Security Attacks on Android Application

Dipakkumar H Patel

Research Scholar

Department ofInformation Technology

Shantilal Shah Engineering College

Bhavnagar, india

Abstract - Android operating system has now a days been emerging operating system. It has a remarkable share in the smartphone and tablet market. Considering the wide range of functionalities provided by smartphones, smartphone-users' privacy and security of their personal information is of vital importance. Android has been the most targeted platform for malware attacks, due to being the most popular platform, open source system and its vulnerable architecture. According to the survey, there's significant threat of attacks and these attacks are really hard to detect, therefore it is considered to be a real threat to smartphone security. Here a method to instrument android application and attack on android security is discussed.

Keywords:*Android, Instrumentation, Security*

1.1: Introduction to the Research Problem

Today's era is the era of technology. Smartphone and mobile industry is one of the most blooming industry in the entire world since last couple of decades. In the earlier era, a human had only three basic needs i.e. food, cloth and shelter but in today's world a common human being has four basic needs i.e. food, cloths, shelter and mobile. In such case, security is the most vulnerable aspect for the smartphone users.

This guide provides detailed descriptions and instructions on the use of Soot, a Java optimization framework. More specifically on how we have used and are using Soot in various projects involving different forms of analysis. Soot is a large framework which can be quite challenging to navigate without getting quickly lost. With this guide, we hope to provide the insight necessary to make that navigation a little more comfortable. The reader is assumed to be familiar with basic static analysis on a level similar to and a firm command of the Java programming language and Java bytecode and dalvik code [4].

1.2: Motivation for the Research Work

As far as android applications are concerned we have two options for instrumentation. One being the modification of the Android software stack and second one being the bytecode instrumentation. Modifying the Android Software stack neither feasible nor fair in this situation as it requires changing the underlying kernel, Dalvik Virtual machine, Android Framework and many more things. Also, this solution would require users to modify their devices some times which is a non-trivial task. If the operating system is modified, then each and every component related has to undergo some sort of modifications which is not easy at all. So we are left only with the second method i.e. performing bytecode instrumentation to change the behaviour of the application.

Bytecode instrumentation can be done with the help of tools which are available globally, Soot is one such tool. Thus in bytecode instrumentation we neither modify the kernel nor some sensitive operating system specific things. Most of all the beauty of bytecode instrumentation is that we can do it manually as well. We can instrument the predefined instructions in the source code of the application directly, only thing we need to know is where the instrumentation instructions are to be placed and how should be their syntax. We will be using Eclipse IDE for creation and instrumentation of my custom made application. Thus after instrumenting these instructions manually in the source code we can change the behaviour of the application at the runtime [4].

1.3: Objective and Scope

The objective of this research work is to provide the method to instrument the android application for security attacks. The aim of this research work is to present a method of instrumenting Android Applications which allows the user to instrument corresponding Android Application. Instrumented code alters the behaviour of the original application thus allowing the attacker to make the application behave the way he wants to. Since the instrumentation code runs as an integrated part of the target application with static analysis approaches.In many cases, the source code of the target application is not available. Therefore, a mechanism for conveniently analysing and instrumenting binary applications is required. In this paper, we will give an overview of the 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.

2.1: Introduction to Android

ThiswordmeansalotinpresentHigh-TechWorld.TodaySmartphoneareknown for itsoperatingsystemwhichisAndroid.Earlierthere isnooptionforoperatingsystems like Androidinmobile,asusualthere areSymbian,javafeaturedoperatingsystemsbut today thingshadchangedalot,everyonewantsaSmartphonewhichisfunctionedon Androidonly.EvenifsomeoneasksmethatwhatSmartphonewouldIshouldbuy?Isuggest themtobuy aonewhichconsistofandroidinitwithlatestversionnomatterwhat'sthe cost. In averyshortspan oftimeandroidcreated areputedplaceinthemarket.Whatisthis *Android*actually?Androidisasoftwareclusterformobiledevicesthatincludesanoperating systemOS,keyapplicationsandmiddleware.TheAndroidSDKprovidesthetoolsandAPIs required tobegindeveloping applications on the Androidplatformusingthe Java programminglanguage.Aboutthedesign,Kernelof AndroidisbasedonLinux kerneland furtherfurnishedby Google.

2.2: Android Architecture

TheAndroidOScanbe referred to as a software stack of different layers, where each layer is a group of severalprogramcomponents. Togetherit includes operating system,middleware and importantapplications.Eachlayerinthearchitectureprovidesdifferentservicestothelayer justaboveit. Wewillexaminethe features of eachlayer in detail.

2.2.1: Linux Kernel

ThebasiclayeristheLinuxkernel.ThewholeAndroidOSisbuiltontopofthe Linux 2.6 Kernelwithsomefurtherarchitecturalchangesmadeby Google. Itis thisLinuxthat interactswiththe hardwareandcontainsalltheessential hardwaredrivers.Driversare programsthatcontrol andcommunicate withthe hardware. For example, consider the Bluetoothfunction. All deviceshas Bluetoothhardware init.Thereforethekernelmust includeaBluetooth driver to communicate with the Bluetooth hardware. TheLinux kernel alsoactsasan abstractionlayerbetweenthehardware andothersoftwarelayers.Android uses the Linux forallits core functionalitysuch as Memory management, process management,networking,security settingsetc.AstheAndroidisbuilt onamostpopular andprovenfoundation,itmadetheportingofAndroidto variety of hardware,a relatively painless task.

2.2.2: Libraries

ThenextlayeristheAndroid'snativelibraries.Itisthislayerthatenablesthedeviceto handledifferenttypesofdata.TheselibrariesarewritteninCorC++languageandare specificfor aparticularhardware.

2.2.3: AndroidRuntime

Android Runtime consistsof Dalvik Virtual machine and CoreJavalibraries.DVMisatypeofJVMusedinandroiddevicestorunappsandisoptimizedforlow processingpowerandlowmemoryenvironments.Unlikethe JVM,theDalvikVirtual Machinedoesn'trun.classfiles,insteaditruns.dexfiles..dexfilesarebuiltfrom.class fileatthetimeofcompilationandprovideshigherefficiency inlowresource

environments.TheDalvikVMallowsmultiple instanceofVirtualmachinetobe createdsimultaneouslyprovidingsecurity,isolation,memory managementand threading support.Itis developed byDan Bornstein of Google.

2.2.4: Application Framework

Thesearetheblocksthatourapplicationsdirectlyinteractswith.Theseprogramsmanage the basicfunctionsof phone like resource management,voice callmanagementetc. Asa developer,youjustconsiderthesearesomebasictoolswithwhichwearebuildingour applications.

3.1 APK Build Process

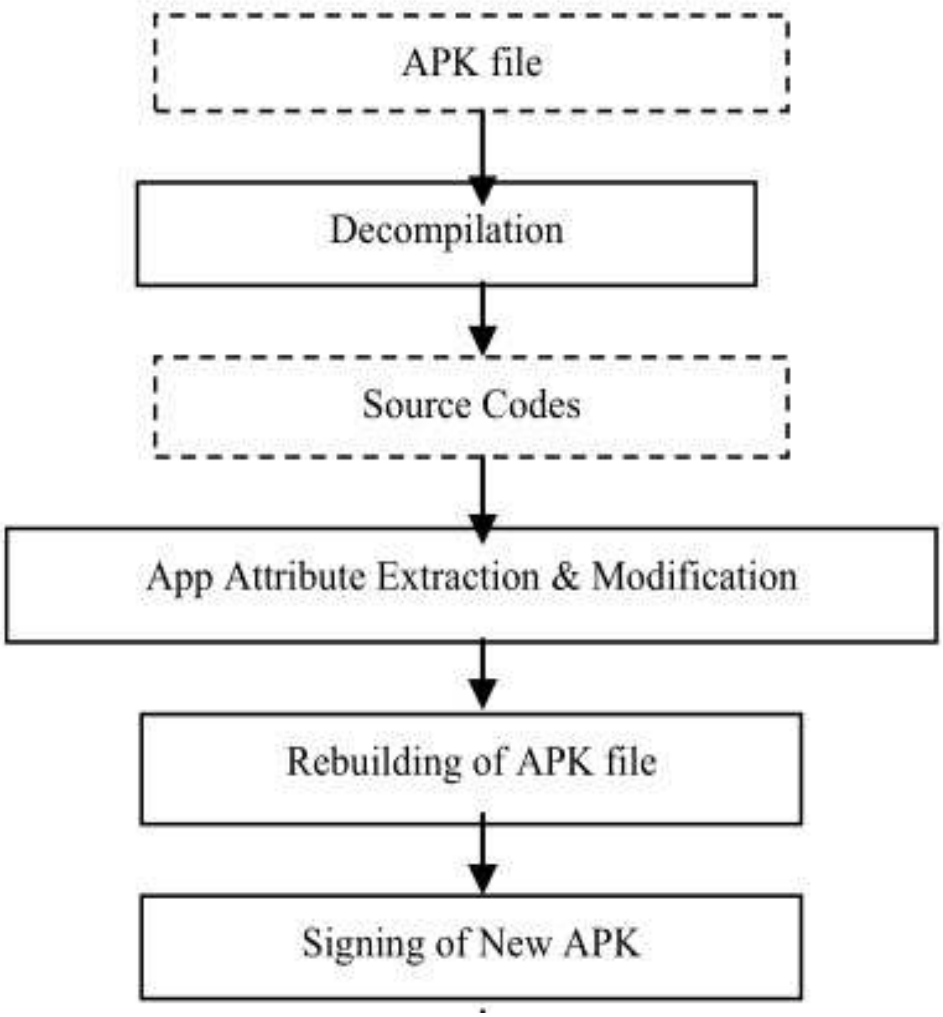

Figure 3.1: APK Build process [4]

3.1.1Smali/Baksmali

Smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. The syntax is loosely based on Jasmin's/Dedexer's syntax, and supports the full functionality of the dex format (annotations, debug info, line info, etc.) [10].

3.1.2Zipalign

Zipalign is an archive alignment tool that provides important optimization to Android application (.apk) files. The purpose is to ensure that all uncompressed data starts with a particular alignment relative to the start of the file. Specifically, it causes all uncompressed data within the .apk, such as images or raw files, to be aligned on 4-byte boundaries [11].

3.3 Soot Framework

Soot is a powerful, extensible, open-source framework for analysing and optimizing Java programs. It enables developers to better understand their Java programs, especially when used as a framework for expressing quick and dirty analyses of Java code. Soot also makes it easy for developers to write code that transforms Java programs. Soot has various intermediate representations mentioned below [14].

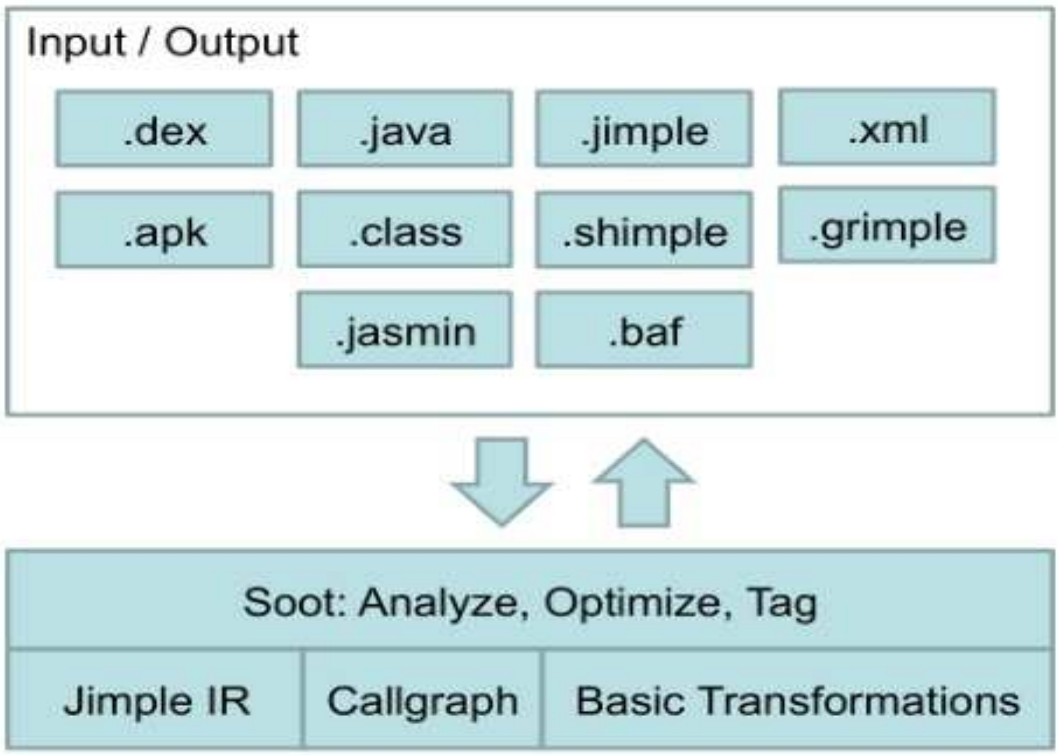

Fig 3.2 Soot Framework [4]

We can use Soot tool in two ways:

1) Soot as a command line tool
2) Soot as an eclipse plugin

1) Requirements to use Soot as a command line tool i.e. as an end-user tool [15]:
   Step-1) Install Java.
   Step-2) Download following 3 jar files from Soot's official site.
   a) polyglotclasses-1.3.5.jar
   b) sootclasses-2.3.0.jar
   c) jasminclasses-2.3.0.jar
   Then put these jar files in your CLASSPATH environmental variable. Then check whether soot is properly configured and working or not by using following commands:

I. **Java soot.Main –help**: It should give you Soot options.
II. **Java soot.Main –version**: It should give you the current version of Soot available on your system. Now Now to generate the Jimple code of a java program, lets create a Test.java file and compile it. We will get Test.class file.

2) If you want to use Soot as eclipse plugin then you can download plugin in eclipse.

3.3.1 Intermediate Representations:

There are 4 types of IR as listed below:

1) Jimple: It is java's simple type-3 address representation. It is stackless representation. It is a principal intermediate representation of Soot
2) Shimple: It is a SSA version of Jimple.
3) Grimp: It is similar to Jimple but with expression aggregated.
4) Baf: It is a compact representation of bytecode.

❖ **Jimple**

Jimple is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation. Jimple representations can be created directly in Soot or based on Java source code(up to and including Java 1.4) and Java bytecode/Java class files(up to and including Java 5).The translation from bytecode to Jimple is performed using a native translation from bytecode to untypedJimple, by introducing new local variables for implicit stack locations and using subroutine elimination to remove jsr instructions. Types are inferred for the local variables in the untypedJimple and added [4].

The Jimple code is cleaned for redundant code like unused variables or assignments. An important step in the transformation to Jimple is the linearization (and naming) of expressions so statements only reference at most 3 local variables or constants. Resulting in a more regular and very convenient representation for performing optimizations. In Jimple an analysis only has to handle the 15 statements in the Jimple representation compared to the more than 200 possible instructions in Java bytecode.

In Jimple, statements correspond to Soot Units and can be used as such. Jimple has 15 statements, the core statements are: NopStmt, IdentityStmt and AssignStmt. Statements for intraprocedural control-flow: IfStmt, Goto-Stmt, TableSwitchStmt(corresponds to the JVM tableswitch instruction) and LookupSwitchStmt(corresponds to the JVM lookupswitch instruction). State-ments for interprocedural control-flow: InvokeStmt, ReturnStmt and Return-VoidStmt. Monitor statements: EnterMonitorStmt and ExitMonitorStmt. The last two are: ThrowStmt, RetStmt (return from a JSR, not created when mak-ingJimple from byte code).

3.4 Proposed Method

Here wehave discussed about current methodologies of instrumenting apk files. We had some introduction to soot framework and its intermediate representation also. Now we are going to work on static instrumentation by using so framework APIs and develop our own driver class, which can be injected to intermediate code and change the behaviour of application.

Here we need to perform following steps to achieve our task.

1) We need to take a target android application.

2) Then unpack the application using soot framework so that we can get intermediate representation.

3) After getting intermediate code we need to analyse the application behaviour and find the vulnerable points for attacks. We can also get call graph of the application to perform the control flow analysis.

4) We will develop driver class that will perform the task of changing behaviour of existing code and then we will inject our payload (code which modifies behaviour of application) into that code.

5) We will now repack this application and load it on simulator or actual device for verification of the instrumentation.

6) If instrumented application works fine then our goal achieved, but if not working as intended then we need to do the same procedure again.

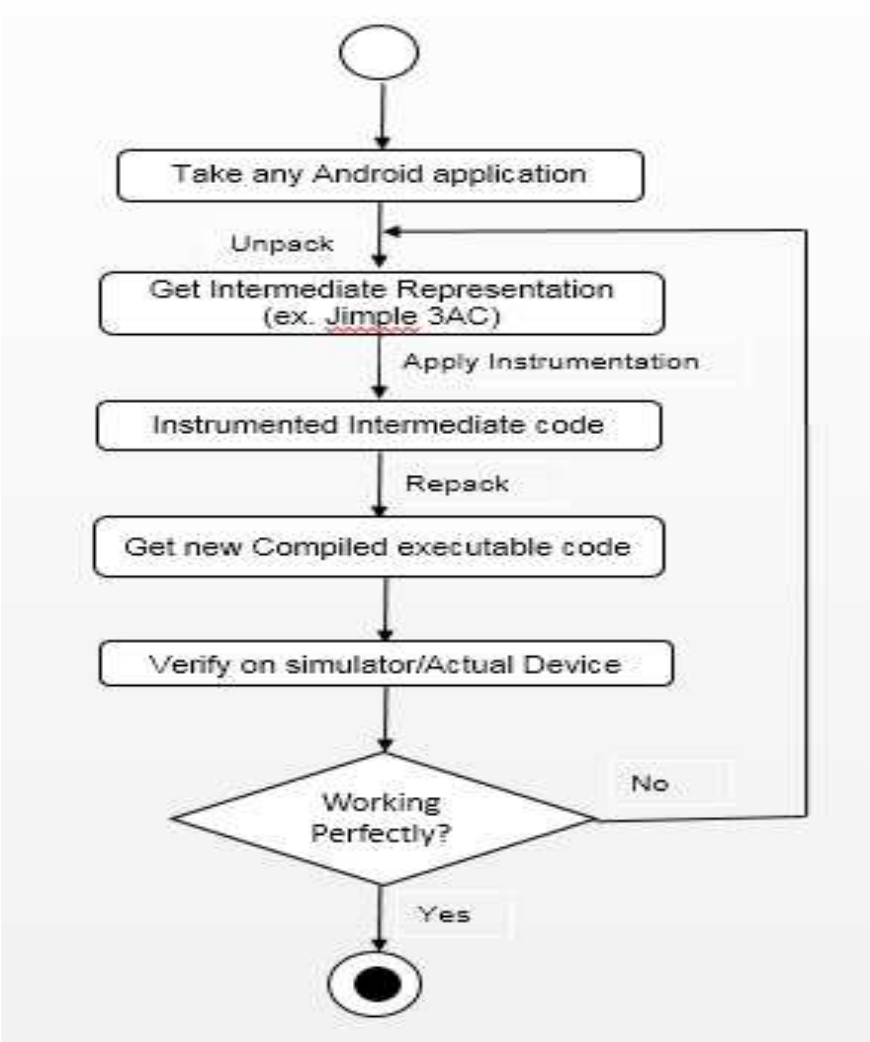

Fig 3.4 Proposed Method

We can instrument this IR code using Disassembler like Smali and insert the trace statements and again recompile it for execution. By using the Android apktool's decode function we need to decompile an Android APK into its constituent parts. It uses baksmali, a disassembler for Dalvik executable (.dex) files used by Dalvik Virtual Machine. Dex files themselves are created from Java class files generated from Java source code. After baksmaling completes, apktool loads the application's binary resource table, which stores or references resources defined by the application, such as xml files containing strings or layouts used by the application. It uses the resource table to decode the application's AndroidManifest.xml, which contains essential information necessary for execution of the app DVM such as the package name, the activities, services, process, and permissions used in the application.

The creation of Jimple was motivated by the difficulty of directly analyzing Java bytecode.Although it is possible to construct a control-flow graph for Java bytecode, the implicit stack masks the flow of data and thus makes the bytecode quite difficult to analyze. In particular, at a given bytecode instructions, it is not at all obvious which previous s 0 produced the stack-based inputs of. Storing data in named local variables, rather than on the implicit stack, makes the local flow of data (along Jimple's Control-Flow Graph) much more obvious. The local variables in Jimple are split according to definition-use chains.

java -cp sootclasses-2.3.0.jar;jasminclasses-2.3.0.jar;polyglotclasses-1.3.5.jar soot.Main - f J Test.class
OR
javasoot.Main -f J Test

The smali code of the Test application:

```
.method protected showResult(Landroid/text/Editable;Landroid/text/Editable;)V
.locals 5
    .param p1, "first"    # Landroid/text/Editable;
    .param p2, "second"    # Landroid/text/Editable;
    .prologue
    .line 47
invoke-interface {p1}, Landroid/text/Editable;->toString()Ljava/lang/String;
move-result-object v3
invoke-static {v3}, Ljava/lang/Float;->parseFloat(Ljava/lang/String;)F
move-result v0
    .line 48
    .local v0, "num1":F
invoke-interface {p2}, Landroid/text/Editable;->toString()Ljava/lang/String;
move-result-object v3
invoke-static {v3}, Ljava/lang/Float;->parseFloat(Ljava/lang/String;)F
move-result v1
    .line 49
    .local v1, "num2":F
add-float v2, v0, v1
    .line 50
    .local v2, "result":F
iget-object v3, p0, Lcom/calculator/Main;->Result:Landroid/widget/TextView;
invoke-static {v2}, Ljava/lang/String;->valueOf(F)Ljava/lang/String;
move-result-object v4
invoke-virtual {v3, v4}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
    .line 51
return-void
  .end method
```

To test our methodology, we took the basic calculator application which adds two numbers. We have instrumented a calculator application using our proposed methodology, which gave us our desired result after modification in the compiled code. We first decompiled the apk file then got the intermediate code of the application. We instrumented the intermediate code and then recompiled the apk file. We need to sign the newly generated apk file before deploying it to actual device or on simulator.

We have instrumented the intermediate representation such that the code which actually performs the multiplication, has been changed to perform the addition. Actual user don't know what happens in the background if you modify the code. This way you can also add some malicious code in the background to perform malicious activities or to get access to unauthorized data of user.

Conclusion:

This research work has presented a method of instrumenting Android Applications which allows the user to instrument corresponding Android Application. Instrumented code alters the behaviour 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.

It also 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 thebehaviour of the original application thus allowing the performer to make the application behave the way he wants to.

Future work:

In future instrumentation of the android application to perform various attacks dynamically can be carried out. We need to develop various drives classes that can perform various kinds of activities. Thus with the help of instrumentation we can develop a whole new era of android application with are fully featured.

References

[1] Andreas Bauer, Jan-ChristophKüster, Gil Vegliach "Runtime Verification meets Android Security" ISBN: 978-3-642-28890-6

[2] Patrice Pominville, Feng Qian, Raja Vall´ee-Rai, Laurie Hendren, and Clark Verbrugge, McGill University "AFramework for Optimizing Java UsingAttributes" June 2013. ISBN: 978-3-540-41861-0

[3] Alexandre Bartel, Jacques Klein, Yves Le Traon "Dexpler: Converting Android DalvikBytecode to Jimple for Static Analysis with Soot" University of Luxembourg-SnT, Luxembourg. ISBN: 978-1-4503-1490-9

[4] StevenArzt, Siegfried Rasthofer and Eric Bodden "InstrumentingAndroid and JavaApplications as easy as abc" Oct 2013 TechnischeUniversitat Darmstadt, Germany. ISBN: 978-3-642-40786-4

[5] Mu Zhang, Heng Yin "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications", Syracuse University. ISBN: 978-1-4614-5522-6

[6] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry "Towards Taming Privilege-Escalation Attacks on Android". ISBN:978-2-87971-107-2

[7] Claudio Marforio, AurélienFrancillon, SrdjanCapkun "Collusion Attack on the Permission Based Security Model and its Implications for Modern Smartphone Systems".ISBN: 978-1-4503-2477-9

[8] Seonho Choi, Michael Bijou, Kun Sun, and Edward Jung "API Tracing Tool for Android-Based Mobile Devices". ISBN: 978-1-4658-2578-6

[9] http://ibotpeaches.github.io/Apktool/

[10] http://code.google.com/p/smali/

[11]http://developer.android.com/tools/help/zipalign.html

[12]https://github.com/pxb1988/dex2jar

[13]http://jd.benow.ca/

[14] https://github.com/Sable/soot

[15]http://www.bodden.de/2013/01/08/sootandroid-instrumentation/