Reverse Engineering Android Applications

by

Enrico Galli

(Under the Direction of Kang Li)

Abstract

Android is the fastest growing mobile platform. This market growth has attracted malware authors. For example, in 2011, the number of malware applications targeting Android more than doubled. However, while the number of malware applications has increased, there has not been a corresponding growth in the number of tools which can be used to analyze Android applications.

In this thesis we attempt to fill some of the holes in reverse engineer's toolkit. To achieve this goal, we opted to provided a formalized process for extracting and packaging Android applications, a novel technique which enables engineers to debug applications at the bytecode-level, and a dynamic taint analysis based approach for correlating inbound with outbound network packets.

Index words: Android, Dalvik, Debugger, Mobile, Reverse Engineering, Security

Reverse Engineering Android Applications

by

Enrico Galli

B.S., The University of Georgia, 2009

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2012

Reverse Engineering Android Applications

by

Enrico Galli

Approved:	
Major Professors:	Kang Li
Committee:	Roberto Perdisci Lakshmish Ramaswamy

Reverse Engineering Android Applications

Enrico Galli

July 15, 2012

Acknowledgments

Graduate school has been a long and interesting experience. I feel that the people that I have met have changed me in positive way. I would be lying if said that it has been an easy and trouble-free ride. However, thanks to those around me, I have somehow managed to make to the end. I would like to thank my committee for the time and dedication that they have spent indulging my crazy ideas. Thanks to my major process Dr. Kang Li for introducing me to Android and going above and beyond the duties of a major professor. Thanks to Dr. Roberto Perdisci and Dr. Lakshmish Ramaswamy for being part of my committee and teaching some of the most interesting classes at UGA. I also would like to thank my labmates and disekt teammates for all the awesome times we have had during capture-the-flag competitions. Finally, I would like to thank my family for keeping me sane this past few years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As of 2011, Android become the largest mobile platform with a market share of 47%[25]. This popularity has attracted malware writers to the platform. The same year, Android experienced an explosive increase in Android malware software[29]. Malware authors see an opportunity to steal large quantities of lucrative informational. Attackers will often steal password, subscriber identity module(SIM) numbers, and phone numbers.

In order to better understand Android malware we have developed a set of tools and techniques for the purpose of understanding and reverse-engineering Android applications. These tools and techniques have been organized in a multi-level hierarchy matching that of the Android platform. Android, as shown in figure 1.1, can be organized into multiple layers: applications, Dalvik virtual machine, and the Linux kernel. While the default programming language of Android is Java, Google opted not to use a standard Java virtual machine. Unlike a standard Java virtual machine, the Dalvik virtual machine(Android's virtual machine) is based on a register architecture. Android also include a heavily modified version of the Linux kernel. This kernel has been patched to include support for Android's security system, IPC(Binder), and power management. Instead of relying on the Dalvik VM, Android use the operating system kernel to enforce its security restrictions [23, 20].

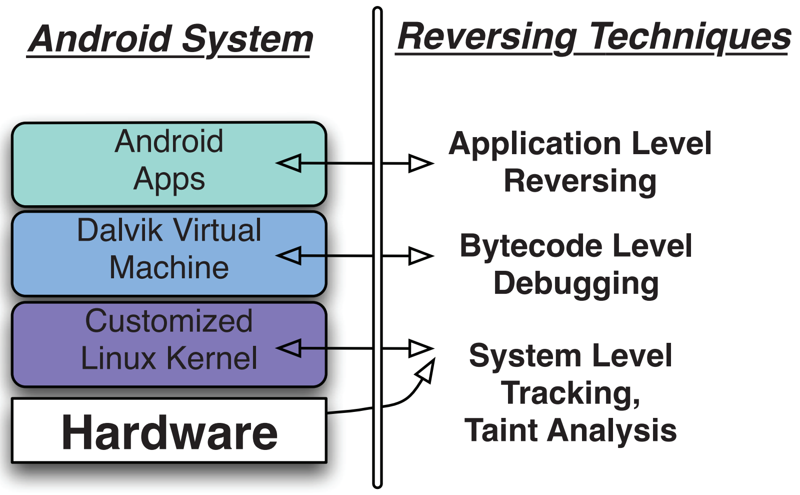


Figure 1.1: Graph showing a broad overview of the Android layers and the techniques we developed to understand them.

In the context of this thesis, reverse-engineering refers to the process of understanding the run-time behavior of a compiled application. Reverse-engineering allows researcher find better ways to defend against malware.

In this thesis we explore ways to extract useful information from each of the Android layers. At the application-level we explore a technique for modify compiled applications. At the bytecode-level, we demonstrate how to enable bytecode-level debugging using existing Java debuggers. Finally, we show a system-level approach for correlating inbound and outbound network packets.

1.1 Rationale for a multi-level reversing framework

Reverse-engineering applications is complex and time consuming process. It requires gaining deep knowledge and understanding of the application's behavior. This thesis attempts to expedite this process. To achieve this goal, we show how to extract information from each of the Android layers. At the application-level, we focus on acquiring static information about the application and modifying the application bytecode. In the virtual machine layer, our bytecode-level debugger shows how the application modifies it internal state at run-time. Finally, our system-level tracker uses a black box approach to extract the network behavior of the application.

1.2 Contribution

This thesis presents the following contributions:

- **A formalized process for reversing Android applications:** In chapter 2, we created a general approach for extracting, decoding, and modifying Android applications without requiring the original source code. This general approach should serve as a starting point for researchers. While each application is different, the general steps we outline will apply to all applications.

- **A way to enable bytecode-level debugging on Android applications:** In chapter 3, we developed a process which allows Java debuggers to operate at the bytecode-level. Once an applications has been altered with our process, a standard Java debuggers is able to set breakpoints on individual bytecode lines and modify the content of the Dalvik registers.

- **NetTaint – a TEMU plug-in for tracking network packets within a system:** In chapter 4, we developed a plug-in that takes a black box approach to extract the network behavior of an executable. The plug-in uses taint tracking techniques to understand how the application responds to network stimulus.

1.3 Outline

Chapter 2 presents and tests a formalizes approach for reversing Android applications.

Chapter 3 covers and tests the steps required to enable bytecode-level debugging on the Dalvik virtual machine.

Chapter 4 explains the details behind our approach to correlate inbound and outbound network packets.

Chapter 5 conclude with a summary of the contributions and avenues for future work.

Chapter 2

Application-level reversing

An important part in the process of analyzing Android applications are the steps required modify Android application packages(APK). Researchers often need to create a safer version of malware applications. Some of the ways in which malware can be nullified include altering the binary's execution flow in order to bypass attack selections and changing addresses of remote hosts. Altering the execution flow allows researchers to safely run the application without jeopardizing other systems. Changing the remote host allows the researchers to capture the attacks used by the application or pretend to be a command-and-control server.

Ideally, altering an application would be performed using a high-level programming language. However, more often than not, it is not possible to access the original source code. Projects, like dex2jar[5] and ded[27], have attempted to translate Dalvik bytecode back into Java bytecode. However, while they show some promise, these project often produce fairly broken code on simple constructs. Figure 2.1 shows one such simple construct and its resulting broken decompiled code. Since the state of tools used to generate Java source code currently do produce valid code, we devised a Dalvik bytecode based approach. This ensures that the encoding stage produces a valid program.

In this chapter we present a formalized process for modifying existing applications. The

```java
class Broken {
  public static void main(String [] args) {
    if(args.length < 2) return;
    System.out.println("Do stuff");
    if("run".equals(args[1])) {
      System.out.println("run!");
      return;
    }
  }
}
```

(a) Original source code.

```java
class Broken {
  public static void main(String[]
      paramArrayOfString) {
    if (paramArrayOfString.length < 2);
    while (true) {
      return;
      System.out.println("Do stuff");
      if (!"run".equals(paramArrayOfString
          [1]))
        continue;
      System.out.println("run!");
    }
  }
}
```

(b) Decompiled source code.

Figure 2.1: Comparison of the original source code and source code from decompilation.

process is split into five stages, extracting, decoding, encoding, and packaging. After describing each state, we show how the process can be used to modify a real malware application. The malware sample was modified to connect to a local server.

2.1 Application alteration pipeline

Our Android application alteration pipeline includes five steps: extraction, decoding, modifying, encoding, and packing. Figure 2.2 shows a rough overview of the process required to modify an existing `.apk` file. The purpose of the extraction and decoding steps is to transform an `.apk` file into an easily editable form. The modification step is an application specific step which involves reading and altering the bytecode. The encoding and packing steps create a new `.apk` file from the altered files.

2.1.1 Extraction

The extraction step involves separating an `.apk` file into multiple files. Since `.apk` files are based on the JAR specification, they can be extracted with any Zip-based compression

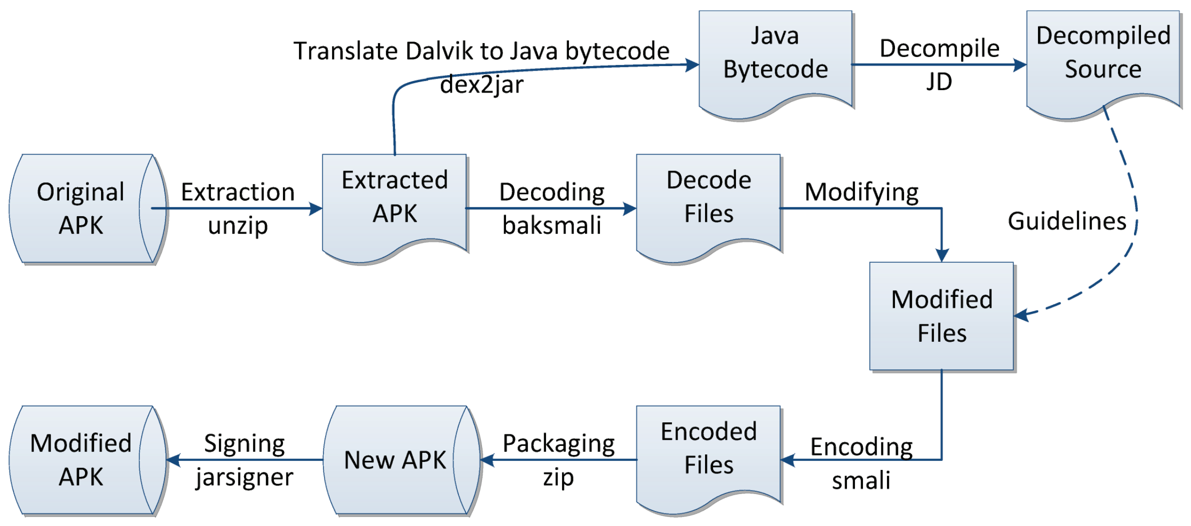


Figure 2.2: Diagram showing an overview of the modification process.

utility. An overview of the APK format is included on appendix A.

As part of the extraction process, the `META-INF` directory must be deleted. This directory contains various files used to verify the integrity of the JAR. Since we will be modifying the contents of the archive, the files inside `META-INF` directory will need to be recreated.

2.1.2 Decoding

The purpose of the decoding step is to create human readable versions of the binary files. Files with a `.dex` extension contain Dalvik bytecode. These Dalvik bytecode files can be converted into an equivalent text format by the Baksmali[12] disassembler. Baksmali disassembles a single `.dex` file into multiple `.smali` files. Each file corresponds to a single Java class.

Android uses a binary XML format to speed the application loading process. Therefore, before any reading or altering can occur, these binary XML files must be converted into an

```
#extract APK in a folder
unzip program.apk -d program
cd program
#remove old signatures
rm -r META-INF
#decoding .dex into smali files
java -jar baksmali.jar classes.dex
#decode XML
java -jar AXMLPrinter2.jar AndroidManifest.xml > DecodedManifest.xml
```

Figure 2.3: Shell script that can be used to extract and decode an APK file.

equivalent text representation. This can be achieved using the AXMLPrinter2[3] utility.

Figure 2.3 shows a simple shell script that can be used to automate the extraction and decoding steps. The example `program.apk` archive is extracted into a directory called `program`. Then, the `classes.dex` is decoded into multiple `.smali` files. By default, Baksmali extracts the source files into a directory called `out`.

2.1.3 Modification

The modification step, as the name implies, involve making changes to the application. This is the application and task specific part of the process. It involves altering the application bytecode, `AndroidManifest.xml`, application assets, and/or resources. To expedite the process, a tool like `dex2jar`, combined with a standard Java decompiler like JD-GUI[10] or JAD[8], can be used to generate Java source code. While the resulting Java code will probably not compile, it can still be used to get a general idea of a program's structure and algorithms.

2.1.4 Encoding

The encoding steps is similar to the decoding step. First, all modified `.xml` files must be covered back into their binary formats. Then, a new `classes.dex` needs to be created from the modified `.smali` files. This steps is performed using the `smali` assembler. The `smali` assembler assembles a directory with all `.smali` files into a single `.dex` file.

2.1.5 Packing

The packing step is based on Android's standard build process[19]. First, all application files, such as the assembled `.dex` files, binary `.xml` files, and application assets, must be stored in a Zip archive.

Android requires that all applications be cryptographically signed with an RSA certificate. Android uses these signatures to verify the integrity and authorship of the archive. The Android installation process will reject any unsigned `.apk` files. The process to sign an `.apk` file is based on the JAR signing process[7]. The `jarsigner` utility is used to sign `.apk` files with RSA certificates. Self-signed certificates are only valid during development.

The final packing step aligns the contents of the `.apk` file on a 32-bit boundary. Zip alignment is performed with the `zipalign` utility. This restructuring allows Android to directly memory-map sections of the archive in order to improve performance. While this step is not required, the official documentation recommend Zip aligning on all `.apk` files.

Figure 2.4 shows a typical encoding and packing script. The script copies all the application resources into a new temporary `build` directory. A zipping utility is used to creates a new `.apk` file from the temporary directory. The archive is then signed with a private RSA certificate. Finally, the signed `.apk` is aligned on a 4-byte boundary with the `zipalign` utility. The script is also able to optionally create a new RSA certificate.

```
#create temporary folder
mkdir build
#copy everything
cp -r * build
rmdir build/build
#rebuild modified classes.dex
rm build/classes.dex
java -jar smali.jar -o build/classes.dex out
#rebuild modified AndroidManifest.xml
rm build/AndroidManifest.xml
AXmlEncode DecodedManifest.xml AndroidManifest.xml
#create apk
cd build
zip -r ../temp.apk *
cd ..
#(optional) keytool -genkey -keyalg rsa -alias MyCert
jarsigner temp.apk MyCert
#aling apk on 32-bit a boundary
zipalign 4 temp.apk build.apk
```

Figure 2.4: Script for extracting and decoding an APK file.

2.2 Evaluation

This section cover the evaluation of our process. We decided to modify a known Android trojan for the purpose of restricting its reporting features.

For our malware sample, we selected a variant of the DroidDream Android trojan. This sample was originally released on the Android Marketplace under the name of “Magic Hypnotic Spiral”. DroidDream infested application attempt to gain root privileges, open a back door for more malware, and transmit sensitive information to a remote server.

In order to test our process, we decided to modify DroidDream’s normal reporting behavior. Instead of connecting to an unknown remote server, the application will report to a known local server.

Finally, we tested the modified application by installing it on an emulator. The emulator state was monitored to validate the success of the application modifications.

2.2.1 About DroidDream

In 2011, multiple applications infested with the DroidDream trojan were released on the Google Play store. These applications were repackaged free application with a trojan payload. Google was forced to quickly remove them from the market [11].

When one of these DroidDream infested application is started, it first attempt to gain root access on the device by using two separate privilege escalation attacks. It should be noted that these techniques do not work on Android devices running version 2.2.2 or higher. After attempting to gain root access, the application will gather information about the device(IMEI/IMSI numbers and the Android version). This information is then sent to a remote HTTP server. All data is encrypted using a simple XOR-based scheme. Finally, the application installs a service called `com.android.providers.downloadsmanager` as a system application. The purpose of this service is to allow the attacker to install more software after the original infection. It will periodically check the same HTTP server for more software[28].

2.2.2 Modifying DroidDream application

As previously stated, the goal of the modifications was to create a safe executable. To achieve this, we followed our modification procedure and changed the address of the HTTP server. We extracted, disassembled, patched, and reassembled the application.

Extracting and decoding

The extracting and decoding steps on DroidDream infested applications is slightly different than on regular applications. DroidDream application contain a nested `.apk` file in the `assets/` directory(under the name `sqlite.db`). Therefore, the extracting and decoding steps must also be performed on the nested `.apk` file.

```
public static void crypt(byte[] buffer) {
  int pos = 0;
  for(int i = 0; i < buffer.length; ++i) {
    buffer[i] = (byte)(buffer[i] ^ KEYVALUE[pos]);
    ++pos;
    if(pos == keylen) {
      pos = 0;
    }
  }
}
```

Figure 2.5: Encryption function used by DroidDream in the main application.

Modifying the bytecode

As previously stated, our goal was to modify the address of the server used by DroidDream. To achieve this goal, we had to locate where the program is storing the address of the remote server. DroidDream applications store the address of the remove server in two locations. In the main application, it is stored on a public static byte array named `com.android.root.Setting.u`. In the downloader service, it also stored in a public static byte array. However, the array is named `com.android.providers.downloadsmanager.a.e.a`. Both of these addresses are encrypted using function shown in figure 2.5. The function XORs the contents of an array with a static key. As with the addresses, there are two copies of the key. In the main application, the key is stored under `com.android.root.adbRoot.KEYVALUE`. In in service, it is stored under `com.android.providers.downloadsmanager.a.a`. We noticed that the main application and the system services are using the same key.

Using the symmetric property of the encryption function, we were able to craft a new byte array for the application and service. For the main application, we used `http://10.0.2.2/main`. For the service, we used `http://10.0.2.2/service`. We opted to use distinct URLs in order to differentiate between the traffic from the main application and the service. The IP address `10.0.2.2` corresponds to address of the Android emulator's host machine.

Once were able to generate the encrypted addresses, we modified the classes' initialization

```
.method static constructor <clinit >()V
  # start of the clinit
  const/16 v0, 0x14                              # new array will be 20
      bytes long "http://10.0.2.2/main".length()
  new-array v0, v0, [B                           # v0 = new byte[0x14]
  const/16 v1, 0x0                               # 0th index
  const/16 v2, 0x5e                              # new value
  aput-byte v2, v0, v1                           # v0[0x0] = 0x5e
  const/16 v1, 0x1
  const/16 v2, 0x2a
  aput-byte v2, v0, v1
  # ...rest of the bytes...
  sput-object v0, Lcom/android/root/Setting;->u:[B  # Setting.u = v0
  # end of the clinit method
  return-void
.end method
```

Figure 2.6: Part of the modified Dalvik bytecode used to populate the address array.

method(<clinit>). Figure 2.6 shows an excerpt from the bytecode we generated to populate the new address. It creates a new array with the size of our modified address(**new-array**), populates it one index at a time(**aput-byte**), and stores it in the static field(**sput-object**). This process was performed on both the main application and the downloader service.

Repackaging modified application

We followed a process similar to figure 2.4 for both the main application and the downloader service. Since main application contains the service as an asset, we encoded and packed the `sqlite.db` APK first. Then, we encoded and packed the main application. We did not modify the `AndroidManifest.xml` from either the main application or `sqlite.db`.

2.2.3 Testing modifications

In order to test our modifications, we installed the application on the Android emulator. Once the application was installed, we started the application from the application menu.

```
POST /main HTTP/1.1
User-Agent: Dalvik/1.1.0 (Linux; U; Android 2.1-update1; sdk Build/ECLAIR)
Host: 10.0.2.2
Accept: *, */*
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 256

<encrypted xml data>
```

Figure 2.7: Captured HTTP headers from the main DroidDream application.

We then proceeded to capture all incoming connections with our local HTTP server. From these captures, we were able to identify connections from the main application.

Environment

We tested our modified application on the official emulator included with the Android software development kit. We selected a system image for the Android platform version 2.1. This version was picked due to its age. Older images tend to perform better on the emulator. This older image is also vulnerable to the privilege escalation exploits included with in the DroidDream sample.

Instead of the utilizing a full HTTP server, we opted to use a simple shell script and Netcat[6]. This choice was selected in order simplify the testing environment. Netcat does not try to understand the data being sent or received. Therefore, it will always accept connections regardless of their validity.

Captured data

After installing the application using `adb`, we started it from the Android application menu. As the application stated, our HTTP server received the contents of figure 2.7.

In addition to the HTTP data, the application was able to successfully gain root access on the system. Figure 2.8 shows a process list after the application gained root access.

```
# ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
<other processes>
app_28    239   30    133056 23700 ffffffff afe0da04 S com.magic.spiral
app_28    245   30    128852 19368 ffffffff afe0da04 S com.magic.spiral:remote
app_28    257   245   728    324   c01537e8 afe0c7dc S /system/bin/sh
root      1009  37    728    324   c003d444 afe0d6ac S /system/bin/sh
root      1272  30    128592 19912 ffffffff afe0da04 S com.magic.spiral:remote2
root      1278  1272  728    324   c01537e8 afe0c7dc S /system/bin/sh
root      1289  1009  868    332   00000000 afe0c7dc R ps
```

Figure 2.8: Partial process list after the DroidDream trojan performed a successful privilege escalation attach.

`com.magic.spiral:remote2` is currently running as root.

Chapter 3

Bytecode-level debugging

Debugging at the bytecode-level is an important part of our multi-level reversing framework. It enables the analysis of the runtime behavior of applications without requiring the original source code. The goals of our bytecode-level debugger were: support for breakpoints on individual bytecode instructions, and read/write access to the Dalvik registers.

The Dalvik virtual machine does not naively support bytecode-level debugging. For example, Dalvik's Java Debugging Wire Protocol(JDWP) does not provide a way to access Dalvik registers directly. Since JDWP was designed to the extensible, Dalvik's debugging support could be extended to include Dalvik specific methods. However, the engineering work required would be nontrivial. This effort would entail modifying Dalvik itself as well as creating a new debugging client. Instead, we opted to create an offline approach that tricks the Dalvik virtual machine into providing the required information through its regular Java debugging channels. This approach also has the added benefit of reusing existing Java debuggers as bytecode-level debuggers.

We propose a novel approach to achieve bytecode-level debugging by leveraging on the existing Java based debugging frameworks. We achieved bytecode-level debugging by modifying the Dalvik binary files, inserting information about the Dalvik bytecode as Java debugging metadata, and tricking existing Java debuggers into using disassembled bytecode files instead of Java source code.

This chapter starts by demonstrating how bytecode metadata can be used to access Dalvik registers and synchronizing bytecode opcodes with the lines on smali files. We also cover some limitations incurred by our approach. Finally, we conclude by demonstrating our trick achieving bytecode-level debugging on simple application.

3.1 Methodology

In order to enable bytecode-level debugging support, we opted to implement a method that modifies bytecode metadata. Our method addresses three challenges:

- Synchronizing debuggers with smali source code.
- Accessing Dalvik registers from a stack based debuggers.
- Maintaining compatibility with existing Java debugging tools.

The metadata we will be modifying is normally used to synchronize the current bytecode with the Java source code, and to gain access to the local variables. However, we will be using it to provide the required information for bytecode level debugging.

Figure 3.1 shows the Java source code which was utilizing to generate the Dalvik bytecode that will be used in this section. This file, named `Hello.java`, was selected due to its simplicity and familiarity to most programmers. A fully JVM and Dalvik disassembled versions can be found on appendix C.1.

3.1.1 Synchronizing debuggers with smali source

Debuggers rely on two pieces of metadata to synchronize bytecode instructions with source files. In the smali format(detail described in appendix B), they are represented by the

```
class Hello {
 public static void main(String[] args) {
  System.out.println("Hello, world!");
 }
}
```

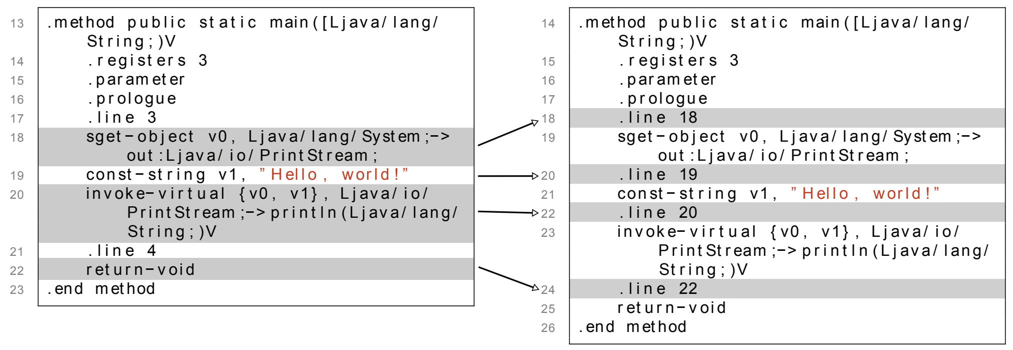
Figure 3.1: Java source from which all the example bytecode for section 3.1.

directives `.source` and `.line`.

The `.source` directive is used to tell the debugger which file is associated with the current class. Therefore, we can just replace the original directive with one that points at a `.smali` file. This will force the Java debugger to load `.smali` files instead of trying to find the original Java source.

The `.line` directive tells the debugger that the following bytecode instructions correspond to a specific line in the Java source. Since the `.source` directive no longer points at the original source, we first strip all the existing `.line` directives. Once all the old `.line` directives have been removed, we add a new `.line` directive for all coding bytecode instruction. These new `.line` directives correspond with the line numbers in the unmodified `.smali` file.

Figure 3.2 shows an example of the process being applied on a file.



```
13 .method public static main([Ljava/lang/
       String;)V
14     .registers 3
15     .parameter
16     .prologue
17     .line 3
18     sget-object v0, Ljava/lang/System;->
           out:Ljava/io/PrintStream;
19     const-string v1, "Hello, world!"
20     invoke-virtual {v0, v1}, Ljava/io/
           PrintStream;->println(Ljava/lang/
           String;)V
21     .line 4
22     return-void
23 .end method
```

```
14 .method public static main([Ljava/lang/
       String;)V
15     .registers 3
16     .parameter
17     .prologue
18     .line 18
19     sget-object v0, Ljava/lang/System;->
           out:Ljava/io/PrintStream;
20     .line 19
21     const-string v1, "Hello, world!"
22     .line 20
23     invoke-virtual {v0, v1}, Ljava/io/
           PrintStream;->println(Ljava/lang/
           String;)V
24     .line 22
25     return-void
26 .end method
```

Figure 3.2: Diagram showing the line renumbering process.

3.1.2 Gaining access to the Dalvik registers

Dalvik uses registers instead of an execution stack. Since the Java Debug Wire Protocol was not designed with a register machine in mind, there is no way to directly access the register. However, we can expose them to the debugger using the local variable directives. Local variable directives are normally used to inform the debugger about the variables available in the current scope. The `.local` directive is used to indicate which registers correspond to which local variables. The `.end local` directive is used to indicate that the register no longer represents a variable. Utilizing these directives, we are able to expose registers as local variables of the same name. When data is stored in a given register, we create a new local variable with the same name as the register. If the variable already exists, and it is of a different type, we use `.end local` to remove it before creating the new one. Figure 3.3 shows a processed method.

```
.method public static main([Ljava/lang/String;)V
    .registers 3
    .parameter
    .prologue
    .line 18
    .local p0, p0:[Ljava/lang/String;
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    .local v0, v0:Ljava/io/PrintStream;
    .line 19
    const-string v1, "Hello, world!"
    .local v1, v1:Ljava/lang/String;
    .line 20
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
        String;)V
    .line 22
    return-void
.end method
```

Figure 3.3: Bytecode after the introduction of the `.local` directives.

3.2 Implementation

To expedite the development, we opted to modify the smali assembler. We added a command-line option that tells smali to alter `.source` directives, strip the old `.line` directives, add new `.line` directives based on the `.smali` file, and create local variables for the Dalvik registers. All of these steps are performed in the assembler's tree walker class. The purpose of the tree walker is to generate a `.dex` file from the parser's output.

3.3 Limitations

While a metadata based approached allowed us to perform bytecode-level debugging, it does have some important limitations. These limitations include:

- **Dynamically loaded bytecode:** This limitations stems from the static nature of our approach. Since we preprocess every class before sending it to the device, we will not be able to step into any code that was missed during the preprocessing phase. The Android framework allows developers to load extra code(both native and bytecode) at runtime. This extra code could conceivably come from external(such as an internet server) sources or obfuscated(packed or encrypted) assets inside the `.apk` file.

  One solution is to intercept new bytecode before it is loaded. Then, we could modify the new bytecode and inject the needed metadata. This can be achieved by modify the framework's Classloaders classes. This method should allow us to still maintain compatibility with existing debuggers. However, this approach also has its problems. For example, the process of transferring the newly processed bytecode from the VM instance to the debugger could cause issues. While the JDWP has an instruction for retrieving bytecode, it was designed to transfer Java bytecode. Therefore, it it not implemented by the Dalvik VM[16].

  Another solution is to extend Dalvik's JDWP implementation with full bytecode support. In the same way that Google extended JDWP with the Dalvik Debug Monitor, an extension which would allow bytecode debugging could be developed. In addition to being able to step into runtime loaded code, this technique would allow us to debug code without needing to preprocess it. However, this technique breaks compatibility with existing debuggers.

Both of the solutions still requires modifying code on the device itself. As many device manufactures lock their device, this could potentially limit which devices could be utilized to debug code.

- **Native code:** While native code debugging was not stated as one of the design goals, it is an important limitation. The Android framework allows developers to include custom native libraries with their applications. Invoking native code is performed through the Java Native Interface(JNI) API. Our metadata only approach is unable to step into the native calls.

  The developers of the Sequoyah project[13] have developed a solution based on the GNU Debugger(`gdb`). It combines the standard Dalvik debugging facilities for the bytecode and an instance of `gdbserver` for the native code.

3.4 Debugger evaluation

In this section we demonstrate the capabilities that our process enables on existing Java debuggers.

3.4.1 Testing environment

We tested our debugging processes on a simple custom built application. This application is a standard hello world styled application. It contains a single button. When the button is pressed, a small message with the words "Hello World!" appears. Figure 3.4 shows the default behavior of the test application after the button has been pressed. We ran our custom

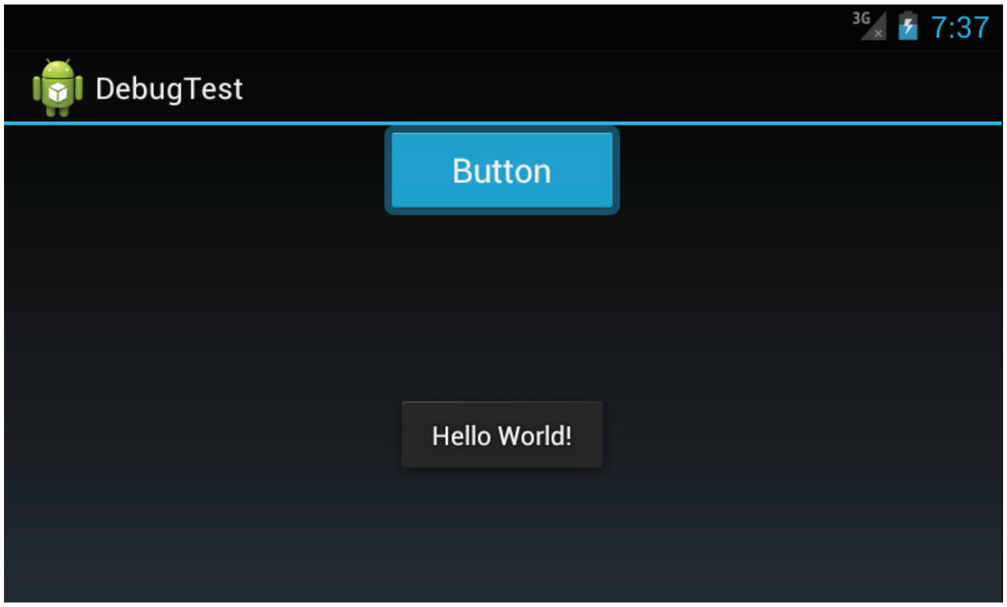


Figure 3.4: Screenshot of the test application running without any modifications.

application on the Android emulator. The emulator was running an Android 4.0.4 system image. We tested the modified application with `jdb`, a command-line debugger included with the Java Development Kit.

3.4.2 Evaluation procedure

We extracted the `.apk` file for our test application. Then, we disassembled the `classes.dex` file with `baksmali`. A new `classes.dex` was created using our modified version of `smali`. The application was repackaged into a new `.apk` file. The new `.apk` was installed on the emulator using `adb`. The application was started from the Android application menu. We connected the Dalvik Debug Monitor Server(DDMS) to the emulator. DDMS is used to forward all debugging ports in an Android system to the local machine. We connected `jdb` to the port matching our test application. We also had to point `jdb`'s source path to

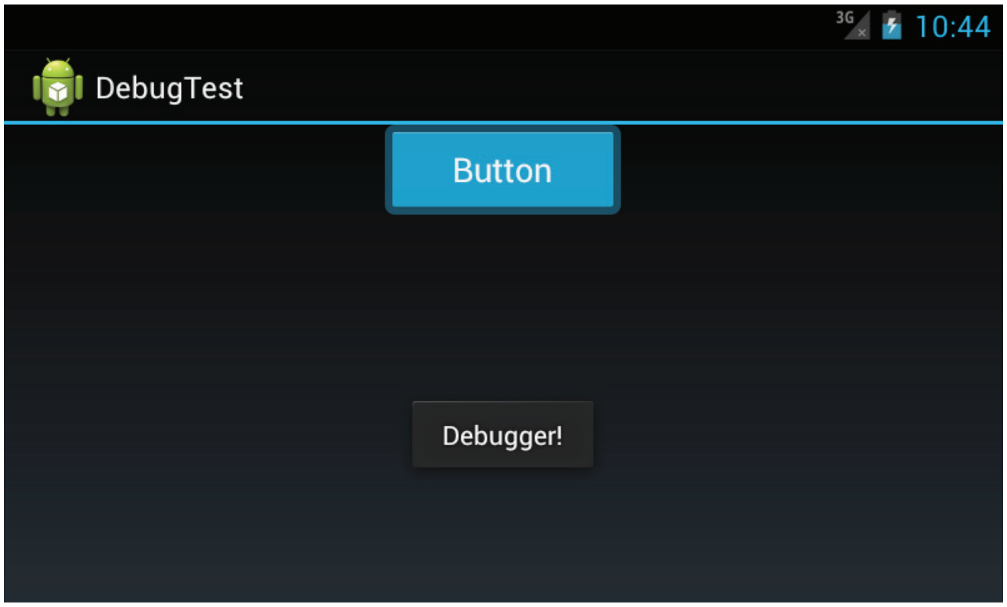


Figure 3.5: Screenshot of the test application running after the debugger changed the contents of the register.

the directory containing our disassembled `classes.dex`. We added a breakpoint on the line `invoke-virtual {v0}, Landroid/widget/Toast;->show()V`. We clicked the button on the Android emulator. When the application hits the breakpoint, we modified the alert text from “Hello World!” to “Debugger!” using the method `Toast.setText`. After the modification was confirmed successful, we resumed execution.

3.4.3 Results

In order to test if metadata changes were successful, we re-disassembled the modified application. We concluded that our process was able to successfully modified the application. Figure 3.6 shows a capture of the steps we took to create a breakpoint and modify the alert text. Figure 3.5 show the result on the Android emulator. With these results we were able to conclude that our process successfully enabled bytecode-level debugging on Android

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,
    port=8700 -sourcepath out
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> stop at edu.uga.DebugTest.DebugTestActivity:37
Set breakpoint edu.uga.DebugTest.DebugTestActivity:37
>
Breakpoint hit: "thread=<1> main", edu.uga.DebugTest.
    DebugTestActivity.buttonClick(), line=37 bci=11
37        invoke-virtual {v0}, Landroid/widget/Toast;->show()V

<1> main[1] print v0.setText("Debugger!")
 v0.setText("Debugger!") = <void value>
<1> main[1] cont
```

Figure 3.6: Captured command used to modify the alert text.

applications.

3.5 Related work

Some community developed tools have attempted to achieve bytecode-level debugging. The most prominent of these tools is called apktool[1]. Apktool wraps existing tools(like the smali assembler and baksmali diassembler) into one executable. However, while it claims to allow step by step debugging, its debugging support has been broken for some time [14]. Another community developed debugging tools goes by the name of AndBug[15]. AndBug was designed as a scriptable debugger for Android. It also support debugging without the original source code. However, AndBug does not support instruction level stepping. It only support breaking on function calls.

Chapter 4

System-level monitoring

As part of our multi-level reversing framework, we developed a method for tracking how the systems responds to network stimulus. This monitoring solution utilizes a dynamic taint analysis framework to track how network packets flow through the system.

This type of system-level tracking is important because it allows researchers to understand how an application responds to network packets. There are multiple circumstance when a researcher would like to know *which packets were generated as a result of an incoming packet*. One such circumstance is when trying to understand command-and-control protocols. Our solution provides enough byte-level propagation information to aid in this task. We named our system **NetTaint**.

In order to test our solution, we designed a test based on the tools included on the Android system image. We used BusyBox to download an HTML file from an Internet server. Our test was able to show that our solution is able to track and graph the flow of network data through the system.

4.1 Related works

In the field of Android application analysis, other projects have attempted to use dynamic taint analysis techniques. One such project is TaintDroid[26]. TaintDroid was created to detect if applications were leaking sensitive information(contacts, phone number, IMSI number, etc.) to the Internet. It utilizes a modified version of the Dalvik virtual machine. Modifying Android's virtual machine has the benefit of higher execution speed. However, if the application uses JNI calls, TaintDroid may not be able to track the flow of taint.

4.2 Overview of dynamic taint analysis

Dynamic taint analysis is a technique designed to track the flow of data through a system. It consists of the following parts:

- **One or more taint sources:** A taint source marks a particular type of incoming data as *tainted*. The system keeps a record of all tainted data in a shadow memory outside of the main system memory. Common taint sources include: command-line arguments, network data(like in NetTaint), and keyboard input.

- **A taint propagator:** The job of the taint propagator is to dictate how taint is copied from one memory location to another. As the system runs, the taint propagator gets invoked when a CPU instruction attempts to operate on tainted data.

- **One or more taint sinks:** Sinks are the destinations for the tainted data. They are used to signal that the tainted data has reached a particular location. In the case of NetTaint, a taint sink is triggered when a packet is about to be sent by the system.

4.3 Approach

System-level monitoring requires precise instrumentation and analysis. To achieve this, our monitoring solution was split into three separate parts. The first part is a system-level taint tracker. Its job is to correlated incoming network packets with outbound packets. The second part groups these packets into TCP half-flows for the purpose of graphing. Finally, we developed a system to generate different kinds of graphs depending on the complexity of the trace being analyzes.

4.3.1 System-level tracking

Leveraging on the work already done by Song et at. on the BitBlaze project [31][21], we were able to develop a simple plug-in(NetTaint) for TEMU. TEMU, a component of the BitBlaze project, is a whole-system emulator built on top of QEMU[30]. It has been designed for dynamic taint analysis. Using TEMU's extendable API, we developed a plug-in that analyzes all data being send and received by a virtual network interface.

As packets arrive from the wire, NetTaint taints the payload of TCP, UDP, and ICMP packets. In order to minimize the noise, while still maintaining a useful set of the information, we decided to only taint the packet payload. All tainted packets are written to an inbound `.pcap` file on the host machine. Each byte on the packet's payload is tainted with an inbound packet number, and its byte offset.

As packets are about to be sent by the guest OS, NetTaint checks the outbound packets for taint information. Unlike the inbound tainter, we do not limit the checking to TCP, UDP, and ICMP packet. In addition to checking any type of packet, we also check the entire packet, not just the payload. This behavior was selected to increase the amount of correlation between packets. In certain situations, a payload might only taint the header of an outgoing packet. One such example are DNS responses. A DNS response would only taint the IP header of an outgoing packet. Once a packet is determined to contain tainted information, it is written to an outbound `.pcap` file on the host machine. In addition to the outbound and inbound `.pcap` files, a separate file containing the relationships between inbound and outbound bytes is written on the host machine. For each tainted byte, we record the outbound packet number, the byte offset of the outbound tainted byte, the inbound packet number, and the byte offset of the inbound byte that tainted the outbound byte.

4.3.2 Rebuilding TCP communication

While dynamic taint analysis is able to match which outbound packets were generated as a result of a give inbound packet, this information is not enough to generate a complete flow graph. Therefore, in order to generate a more complete graph, we devised an algorithm for matching inbound TCP response with outbound TCP request.

When trying to match inbound to outbound packets, we first grouped all inbound packets into TCP half-flows. We defined a TCP half-flow as all the packets that were transmitted from one IP address and port pair to another IP address and port pair. Then, for each outbound packet, we find the corresponding inbound TCP half-flow. Once we have found a matching inbound TCP half-flows, we search for an inbound packet that has arrived at a future time with a TCP acknowledgment number that is greater than or equal to the sequence number of the outbound packet.

4.3.3 Creating a flow graph

Utilizing the files generated by the TEMU plug-in, and some of the information on the TCP headers, we developed three different types of graphs to visualize the flow of network data. We named these graphs: taint tables, packet-level half-flows graph, and half-flows graph. Figure 4.1 contains the key for the packet-level half-flows and half-flows graphs.

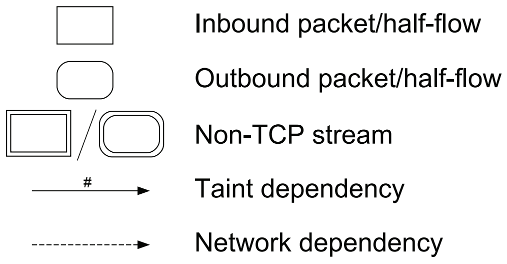


Figure 4.1: Key for packet-level half-flows and half-flows graphs.

Taint tables

Taint tables are a way to directly represent the correlation between all incoming and outgoing tainted network data. Due to the amount of information that these tables need to visualize, we have split the data into two separate graphs. The first graph shows the packet level correlation between all incoming packets and all the tainted packets. The amount of tainted bytes within each packet intersection determines the color used by the graph. A dark shade of blue is used when there is a high correlation between the tainted bytes. Therefore, a given inbound packet heavily influenced the outbound packet. Inversely, lighter shades of blue represent lightly influenced outbound packets. The second graph type shows the byte-level correlation for a given packet pair. Rather than attempting to graph the entire trace, which would provide too much information to be useful, we select a single inbound and outbound packet pair. In this graph, the color represents the location of the bytes inside the outbound packet. Dark red represents bytes in the IP, ICMP, TCP, or UDP headers. Blue represents bytes outside of the headers.

Packet-level half-flows graph

The packet-level half-flows graph shows packet dependencies between half-flows. Dependencies between inbound and outbound packets(solid lines) are generated using the tainted bytes relationships. However, the dependencies between the outbound and inbound packets(dashed lines) are generated from the rebuilt TCP communications. TCP packets are also grouped together into half-flows. Inbound packets are represented by boxes and outbound packets are represented by rounded boxes.

Half-flows graph

The half-flows graph represents the same information as the packet-level half-flows graph. However, it has been simplified to show only half-flow dependencies. Single UDP and ICMP packets are drawn with double borders.

4.4 Limitations

The systems described in this chapter suffer from the following limitations:

- **Only x86 system emulation is supposed:** TEMU only supports x86 based systems. While QEMU supports more architectures than x86, the TEMU team decided to only focus on x86. A non-trivial amount of work would be required to port TEMU to other architectures.

- **UDP and ICMP do not get grouped into half-flows:** As discuss on Section 4.3.2, the current implementation of the visualizer is only able to find network-level dependencies between TCP packets. Therefore, it misses any UDP and ICMP dependencies. This limitation causes gaps on both the packet-level half-flows and half-flows graphs.

  While the current approach being utilizing for TCP would be unfeasible for UDP and ICMP packets, other methods could be attempted. Some ICMP packets, such as ping, could be matched. Deep packet inspection for certain UDP packets, such as DNS, could also be attempted.

- **Graph complexity for non-trivial traces:** While the graphing techniques described in Section 4.3.3 are able to present relatively useful information, their usefulness drastically decreases as the size of the trace increase.

4.5 Evaluation

In order to test the taint propagation system, we download a file from a remote HTTP server using the wget module provide by BusyBox[4]. The test was designed to see if NetTaint was able to trace the flow of tainted bytes across the system. Wget was selected because it has a known dependency profile.

4.5.1 Environment

We ran our experiment on an Android image from the Android-x86 project[2]. The Android version 3.2 was selected since it was the most stable image we could find. We utilized version *1.16.2androidfull* of BusyBox which is included with the Android image.

4.5.2 Experiment

For the experiment, we told BusyBox's wget to download `http://cs.uga.edu/~galli`. This caused BusyBox to send three different types of requests to the network, a DNS request for the IP address of `cs.uga.edu`, an HTTP GET request for the file `/~galli`, and a redirected HTTP GET request for the object `/~galli/`. In order to minimize the amount of traffic, the contents of the final HTML page were forced to be empty. We selected this URL because it allowed us to see how outgoing packets were tainted by both UDP and TCP traffic.

4.5.3 Results

Table 4.1 shows which packets were received and which tainted packets where sent by the system. There were 11 inbound packets(grouped into 3 half-flows). These inbound packets tainted 11 outbound packets(grouped into 2 half-flows).

Figure 4.2a shows the dependencies between output(y axis) and input(x axis) packets. Specifically, all the outbound packets depended on the 0th input packet, which is a DNS response for `cs.uga.edu`. This is because all outbound packets are targeting the same destination IP address. Similarly, outgoing packets 5 to 10 were tainted by the 3rd inbound packet, which contains the URL of the HTTP redirect. Outbound packets 5 to 10 are the HTTP packets for the redirected request.

We can also use the collected data to extract the byte-level correlation in the inbound and outbound traffic. For example, if we examine the byte-level correlation between the 3rd inbound packet(HTTP 301 Moved Permanently) and 7th outbound packet(HTTP GET to the redirected object), figure 4.2b, we see a more detailed view of the influence. The result shows that inbound bytes for `cs.uga.edu` and `~galli/` tainted a large percentage of the outbound data and the TCP checksum. The exact byte correlation in this example depends on the internal code of BusyBox's wget implementation.

Figure 4.2c and 4.2d show how NetTaint can be be used to find the dependencies between network half-flows. Figure 4.2d presents the dependencies between the related half-flows. As seen in the packet-level taint table, the DNS responses(In 0) clearly influenced all outbound half-flows. Similarly, the redirected HTTP request(Out 1) depends on the response(In 1) of the first HTTP request(Out 0).

Figure 4.2c shows the same flow dependency graph but at the packets-level. Packet-level half-flows graphs can show a more detailed picture of the traffic. For example, the redirected HTTP half-flow(Out 1) depended only on the 3rd packet of the first HTTP response(In 1). When inspecting the 3rd packet, we find that it contains the target URL for the redirection.

Examples like the one above show the importance of the packet-level half-flow graph when attempting to pinpoint the exact content that defined the correlation between half-flows.

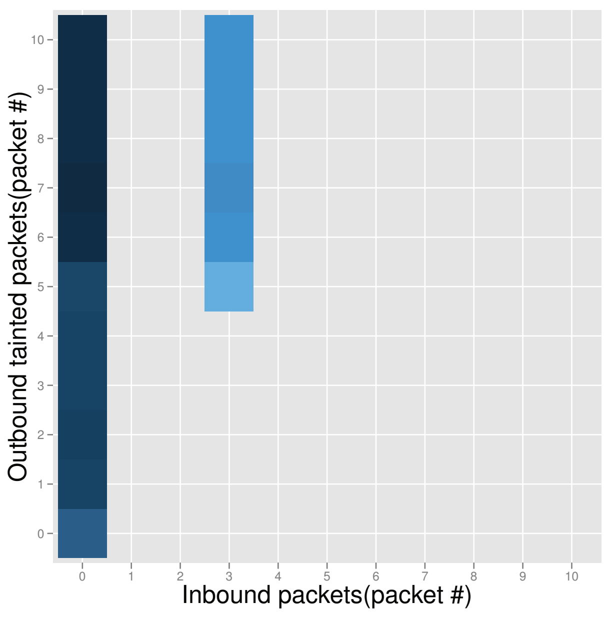
Packet	Half-flow	Description
0	In 0	DNS response
1	In 1	TCP [SYN, ACK]
2	In 1	TCP [ACK]
3	In 1	HTTP/1.1 301
4	In 1	TCP [FIN, ACK]
5	In 1	TCP [ACK]
6	In 2	TCP [SYN, ACK]
7	In 2	TCP [ACK]
8	In 2	HTTP/1.1 200
9	In 2	TCP [FIN, ACK]
10	In 2	TCP [ACK]

(a) List of inbound packets.

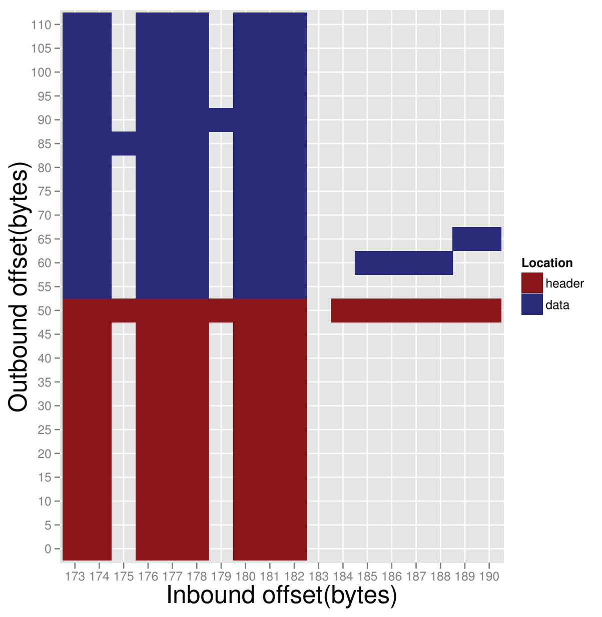
Packet	Half-flow	Description
0	Out 0	TCP [SYN]
1	Out 0	TCP [ACK]
2	Out 0	GET /~galli HTTP/1.1
3	Out 0	TCP [ACK]
4	Out 0	TCP [FIN, ACK]
5	Out 1	TCP [SYN]
6	Out 1	TCP [ACK]
7	Out 1	GET /~galli/ HTTP/1.1
8	Out 1	TCP [ACK]
9	Out 1	TCP [ACK]
10	Out 1	TCP [FIN, ACK]

(b) List of tainted outbound packets.

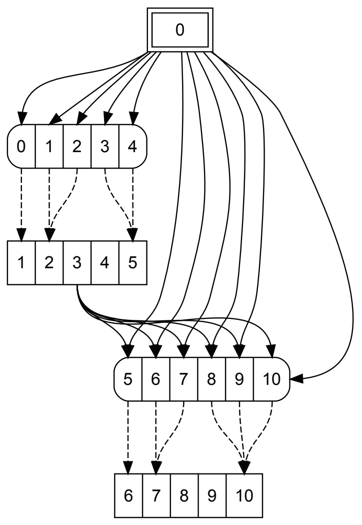
Table 4.1: List of all inbound and tainted outbound packets used in the graphs.



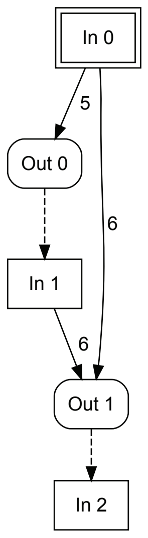
(a) Packets taint table



(b) Bytes taint table for the 3rd inbound packet and the 7th inbound packet



(c) Packet-level half-flows graph



(d) Half-flows graph

Figure 4.2: Graphs generated from a trace of wget downloading a HTTP file.

Chapter 5

Conclusion

This thesis provides multiple debugging and reversing techniques that target the levels of the Android platform. These techniques allow researchers to extract, modify, debug at the bytecode-level, and analyze the network behavior of Android applications.

5.1 Summary of thesis

In chapter 2 we covered our formalized approach for extracting, modifying, and packaging Android applications. This approach expedites the process of revering application by giving researchers a known starting position. Once we described the steps required to extract application, we tested it on a sample of malware. We were able to successfully change the address of its reporting server.

In chapter 3 we outlined the steps required to trick Java debuggers into providing bytecode-level support. This trick allows researchers to understand and modify the run-time behavior of applications. We also tested the our trick on a simple application.

In chapter 4 we showed how, utilizing dynamic taint analysis, we were able to determine how inbound packets influenced outbound packets. We then proceeded to test our approach by tracing the behavior of a known executable.

5.2 Future work

In this thesis we presented techniques for analyzing and debugging Android applications. However, due to time constraints, we were unable to explore certain sections. The following are some ideas for future work:

- **Integration of level:** Currently our framework is comprised of multiple separate tools. In the future we hope to provide a unified solution. This should further expedite the process of reversing Android applications.

- **IDE integration:** In the future, we hope to develop an Eclipse plug-in in the spirit of the Android Development Tools(Google's Android development plug-in for Eclipse) but targeting bytecode rather than Java source. The purpose of this plug-in would be to bring automation to the extraction process. Currently, many of the steps require that we manually run command-line tools after each modification. In addition to automation, the plug-in would bring better debugger integration, syntax highlighting, and refactoring support to smali files.

- **Dynamically loaded bytecode:** As explained in section 3.3, we currently can not step into dynamically loaded bytecode. As it is very likely that malware authors will attempt to use this technique to combat analysis, it is in our best interest to explore ways to achieve debugging on run-time loaded bytecode. Of the solutions that we presented, we believe that modifying Dalvik to support Dalvik specific JDWP verbs is the most promising solution. This solution has the added benefits of removing the preprocessing stage. However, due to the intrusive nature of this approach, it will limit the devices which we could use for testing.

- **Upgrade TEMU:** TEMU, the framework for dynamic taint analysis we used in chapter 4, is currently based on an old version of QEMU. This version of QEMU utilizes a slower and less compatible recompiler. Upgrading TEMU to a new version should increase our NetTaint's performance as well as its compatibility.

Appendix A

Application package file

Application package files, or APK files, are Android's way of storing and distributing applications. The Android Market currently requires that all APKs be under 50MB. However, as for March 2012, applications may provide expansion files for up to 4GB of additional data [18]. The APK format itself is based upon Java's JAR(or **J**ava **AR**chive) format. Figure A.1 shows some of the more common files and directories inside APKs.

META-INF/ APK files, like JARs, are just regular Zip archives with a special `META-INF/` directory. The purpose of this directory is to ensure the integrity of the archive. To achieve this goal, there is a `.SF` file that contains SHA-1 digests for all resources. In addition to these cryptographically secure hashes, the `META-INF/` directory has a `.RSA` certificate for the application. The purpose of the certificate is to ensure the authorship of the archive. All APKs archives must be signed. Otherwise, the system installer will reject the ARK during the install process.

res/ In addition to the `META-INF/` directory, APKs also contain a `res` directory. This directory contains the application's resources. Each resource types is split into separate directories. For example, UI layouts are stored under `res/layout/`. All these resources are

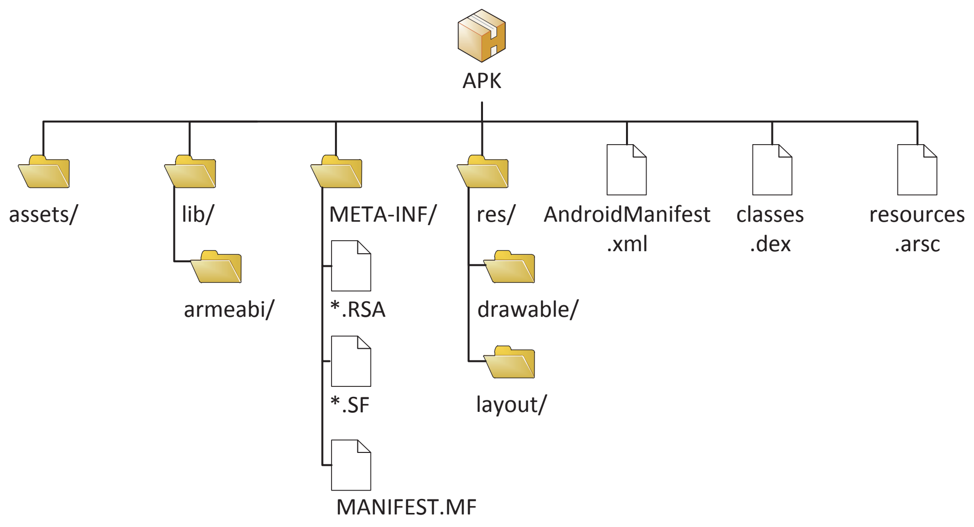


Figure A.1: Common directory structure of APK files.

indexed in the `resources.arsc` file.

assets/ The idea behind assets and resources is similar. They are both non-code resources used by the application. However, assets are less restrictive on the type and size. Also, unlike resources, assets are operated upon using the standard InputStream class.

lib/ The `lib/` directory was added as part of the Android NDK(or native development kit). Architecture specific native code is compiled and stored under separate subdirectories on `lib/`. For example, ARM specific libraries will be stored under `lib/armeabi/`. These libraries will be later loaded by the Dalvik VM and invoked through JNI.

classes.dex The `classes.dex` contains the Dalvik bytecode for the application [24].

AndroidManifest.xml The purpose of the `AndroidManifest.xml` file to describe the application to the Android operating system and system installer. Some of the information it contains includes[17]:

- The Java package name that uniquely identifies the application.
- Description of all the components in the application(i.e. their names and how they interact with the system, other applications, and each other).
- Which of the protected APIs, such as telephony and location services, the application will be using(permissions).
- Permissions that other applications must have in order to interact with this application.
- During development and testing, it can include a list of Instrumentation classes that provide profiling information.
- Minimum API level that the device must support in order to run the application.
- Libraries that the application must be linked against. This does not include the libraries under `lib/`.

Appendix B

Smali reference

This appendix contains a partial reference for the smali language.

Smali format

Smali files are the human readable representation of the Dalvik bytecode instructions and the DEX format. These files are utilizes by the `smali` assembler and the `baksmali` diassembler [12]. The smali syntax is loosely based on Jasmin's syntax[9]. Jasmin is an assembler for Java bytecode. Due to the proximity of Dalvik and Java bytecode, a Jasmin inspired syntax solved multiple problems. Each line in a smali file can be classified into one of the following types:

- **Comments:** As with comments in other programming languages, these constructs allow programmers to embed notes into the source code. Smali only supports single-line comments. They start with a # sign, and end on a newline.
- **Directives:** These define metadata about the class and its instructions. Directives are used to declare fields, methods and the class itself. Appendix B.2 contains a skeleton

smali file. In addition to defining the structure of classes, directives also declare Java annotation and debugging information.

- **Instruction:** These are the bytecode opcodes. They express the program logic itself. A mostly complete reference of the opcodes can be found under `<Android source>` `/dalvik/docs/dalvik-bytecode.html`.

B.1 Date types

This sections contains a list of all the allowed data types.

Mnemonic	Type
V	Void
Z	Boolean
B	Byte
S	Short
C	Char
I	Int
J	Long(64 bits)
F	Float
D	Double(64 bits)
Lpackage/path/ClassName;	Object(type name delimited by L and ;)
[<type>	Arrays(add ['s for each dimension)

Table B.1: List of allowed types and their bytecode mnemonics.

B.2 Class skeleton

Smali class skeleton.

```
.class LSkeletonClass;
.super Lpath/to/BaseClass;
.source "Skeleton.java"

```

```
# static fields
# Declare instance fields with the .field directive
# For example, .field private static final CONST_NUM:I = 0x7b

# instance fields
# Declare instance fields with the .field directive
# For example, .field private num:I

# direct methods
.method constructor <init>()V
  .registers 1 ;at least 1 register for p0(which holds this)
  .prologue
  invoke-direct {p0}, Lpath/to/BaseClass;-><init>()V

  ; initialize fields

  return-void
.end method

.method public static main([Ljava/lang/String;)V
  .registers 1 ;at least 1 register for p0(which hold the first argument)
  .parameter
  .prologue

  return-void
.end method


# virtual methods
# Example method
.method public toString()Ljava/lang/String;
  .registers 1 at least 1 register for p0(which holds this)
  .prologue
  ; custom toString() code
.end method
```

Dalvik bytecode

While Dalvik bytecode still retains some similarities to Java bytecode, the designing decision to utilize a register-based system(Dalvik) rather than a stack-based system(Java) has influenced the bytecode. Rather than using an operand stack, Dalvik holds all temporary values in up to 256 registers. It should be noted that some opcode can only operate on the first 16 registers. The total amount of registers that a given method can use must set at creation time. Each register holds a 32-bit value. For 64-bit values, such as long and double, adjacent registers must be used [22].

Appendix C

Disassembled sample source

This appendix contains the entire disassembled bytecode from the example code. Since disassembled files can be quite long, the entire files were omitted from the main text. We also included the disassembled Java bytecode as a reference.

C.1 Hello.java

This section contains bytecode for the example used in section 3.1.

Disassembled Hello.java file(Dalvik bytecode).

```
.class LHello;
.super Ljava/lang/Object;
.source "Hello.java"

.method constructor <init>()V
    .registers 1
    .prologue
    .line 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 3
    .parameter
    .prologue
    .line 3
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string v1, "Hello, world!"
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;
        )V
    .line 4
    return-void
.end method
```

Disassembled Hello.java file(Java bytecode).

```
; Hello.j

.bytecode 50.0
.source Hello.java
.class Hello
.super java/lang/Object

.method <init>()V
  .limit stack 1
  .limit locals 1
  .line 1
  0: aload_0
  1: invokespecial java/lang/Object/<init>()V
  4: return
.end method

.method public static main([Ljava/lang/String;)V
  .limit stack 2
  .limit locals 1
  .line 3
  0: getstatic java/lang/System/out Ljava/io/PrintStream;
  3: ldc "Hello, world!"
  5: invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  .line 4
  8: return
.end method
```

Bibliography

[1] android-apktool - a tool for reverse engineering android apk files - google project hosting. `https://code.google.com/p/android-apktool/`.

[2] Android-x86 - porting android to x86. `http://www.android-x86.org/`.

[3] android4me - j2me port of google's android - google project hosting. `https://code.google.com/p/android4me/downloads/list`.

[4] Busybox: The swiss army knife of embedded linux. `http://www.busybox.net/`.

[5] dex2jar - tools to work with android .dex and java .class files - google project hosting. `https://code.google.com/p/dex2jar/`.

[6] The gnu netcat – official homepage. `http://netcat.sourceforge.net/`.

[7] How to sign applets using rsa-signed certificates. `http://docs.oracle.com/javase/1.5.0/docs/guide/plugin/developer_guide/rsa_signing.html`.

[8] Jad java decompiler download mirror | tomas varaneckas. `http://www.varaneckas.com/jad`.

[9] Jasmin home page. `http://jasmin.sourceforge.net/`.

[10] Jd|java decompiler. `http://java.decompiler.free.fr/`.

[11] The official lookout blog | update: Security alert: Droiddream malware found in official android market. `http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/`.

[12] smali - an assembler/disassembler for android's dex format - google project hosting. `https://code.google.com/p/smali/`.

[13] A step-by-step guide for debugging native code, by carlos souto. `http://www.eclipse.org/sequoyah/documentation/native_debug.php`.

[14] android-apktool - instructions on how to run debugger for decoded apk. - a tool for reverse engineering android apk files - google project hosting, September 2010. `https://code.google.com/p/android-apktool/wiki/SmaliDebugging`.

[15] Andbug – a scriptable android debugger, December 2011. `https://github.com/swdunlop/AndBug/blob/master/README.rst`.

[16] Handle messages from debugger, May 2011. `<android source>/dalvik/vm/jdwp/JdwpHandler.cpp`.

[17] The androidmanifest.xml file | android developers, March 2012. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[18] Apk expansion files | android developers, March 2012. `http://developer.android.com/guide/market/expansion-files.html`.

[19] Building and running | android developers, April 2012. `http://developer.android.com/guide/developing/building/index.html`.

[20] What is android? | android developers, February 2012. `http://developer.android.com/guide/basics/what-is-android.html`.

[21] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[22] Bornstein, D. Dalvik vm internals. In *Google I/O* (2008).

[23] Brady, P. Anatomy & physiology of an android. In *Google I/O* (2008).

[24] Chen, J. Building an android application 101. In *Google I/O* (2008).

[25] comScore Inc. *2012 Mobile Future in Focus*, February 2012.

[26] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: an information-flow tracking system for realtime privacymonitoring on smartphones. In *Proceedings of OSDI 2010*.

[27] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of android application security. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 21–21.

[28] Maslennikov, D. Malware in the android market, part 2 - securelist, March 2011. `http://www.securelist.com/en/blog/11198/Malware_in_the_Android_Market_part_2`.

[29] Maslennikov, D. Mobile malware evolution, part 5 - securelist, February 2012. `http://www.securelist.com/en/analysis/204792222/Mobile_Malware_Evolution_Part_5`.

[30] QEMU: Open source processor emulator. `http://www.qemu.org/`.

[31] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International*

Conference on Information Systems Security. Keynote invited paper. (Hyderabad, India, Dec. 2008).