Android Application Obfuscation

SBA Research

June 20, 2013

This work has been carried out within the scope of *u'smile*, the Josef Ressel Center for User-Friendly Secure Mobile Environments. We gratefully acknowledge funding and support by the Christian Doppler Gesellschaft, A1 Telekom Austria AG, Drei-Banken-EDV GmbH, LG Nexera Business Solutions AG, and NXP Semiconductors Austria GmbH.

Contents

1 Introduction

Software obfuscation has a long history in the world of desktop computing. A variety of different techniques has evolved to protect code and sensitive information. However, these techniques can often not directly be applied to mobile applications such as those running on Android, at least not at the Dalvik VM level. Therefore, the obfuscation used in most Android applications is not as advanced as the techniques usually found in desktop applications. Obfuscation of software is used by application developers as well as malware authors. There are various different intentions for developers to obfuscate their code. Application authors usually want to protect their intellectual property. Therefore they use obfuscation techniques to prevent analysis of their program. This could be to protect sensitive data as well as prevent removal of license checks or repacking their applications with malicious code included. Malware authors on the other hand want to protect their code from analysis by security analysts or automated scanners such that the intent of the application can not be detected easily.

2 Android Application Obfuscation

Android applications are usually written in Java and executed on a modified Java virtual machine, the Dalvik VM. Thus, existing Java obfuscation techniques such as identifier mangling can often be translated to the Android domain. However, Android not only allows for execution of Dalvik byte code, but developers are also allowed to run native code directly on the processor. This allows for further obfuscation techniques which could not be carried out inside the Dalvik VM alone. The following sections will give a short overview of different techniques used to obfuscate Android applications. Furthermore we will introduce existing tools that can be used to apply some of the presented techniques automatically to existing applications.

2.1 Obfuscation techniques

In essence, there are two separate forms of obfuscation: Dalvik code obfuscation and native code obfuscation. Obfuscation on the dalvik code level is easier to perform, but has only a limited set of abilities compared to native code obfuscation techniques. We will now give a brief overview on available obfuscation techniques for Android applications.

2.1.1 Identifier renaming

One of the easiest methods for obfuscation of Android applications is identifier renaming. The problem with Android, and Java applications in general, is that they contain a vast amount of information about the binary. For example, without obfuscation techniques it is possible to reconstruct the original source code, including variable and function names, from a packed application. Since function and variable names usually describe their intended behaviour, this makes it easy for an analyst to extract information about the application. Listing 5 gives an example for Java source code without any obfuscation applied.

```
1 public class Base64{
2     public String decode( String input )
3     { ... }
4     public String encode( String input )
5     { ... }
6 }
```

Listing 1: Java source code

However, this leakage of information can easily be prevented by mangling function and variable names. An example for this technique can be seen in listing 2. The code has the same structure as the source in listing 5, but with renamed identifiers. While it was easy in the original version to get an idea about the codes intended behaviour, in the obfuscated version it is necessary to further analyse the code to extract this information.

```
1 public class a{
2     public String a( String a )
```

```
3       { ... }
4       public String b( String a )
5       { ... }
6   }
```

Listing 2: Java Source Code with rewritten identifiers

2.1.2 Junk byte insertion

Inserting junk bytes is an easy method for the software author to complicate the analysis of the binary. Although this is a relatively simple way to obfuscate a binary, at least two assumptions have to be considered [12]:
First, the instructions have to be incorrect in a specific way, namely incomplete. This produces a red herring for disassemblers. The second assumption is implied by the first one: The incomplete (junk) instructions must never be reached during execution. If the program tries to execute these instructions it would crash in most cases.
This execution is preventable by using, e.g. an unconditional jump before the inserted junk instructions or a conditional jump if the result is known and predictable, causing the junk code to be jumped over at runtime [14].
Patrick Schulz from dexlabs[1] analysed various disassemblers and reverse engineering tools on their performance when presented with a file with junk bytes inserted.

0003bc:	1250		0000:	**const** / 4 v0, #**int** 5
0003be:	2900	0400	0001:	goto / 16 0005
0003c2:	0001		0003:	<junkbytes>
0003c4:	0000		0004:	<junkbytes>
0003c6:	d800	000	0005:	**add-int** / lit8 v0, v0, #**int** 1
0003ca:	0f00		0007:	**return** v0

Table 1: Disassembly with detection of junkbytes [14]

[1] http://dexlabs.org/blog/bytecode-obfuscation

0003bc:	1250		| 0000:	**const** / 4 v0, #**int** 5
0003be:	2900	0400	| 0001:	goto / 16 0005
0003c2:	0001	0000 d800 0001	| 0003:	dummy-function
0003ca:	0f00		| 0007:	**return** v0

Table 2: Linear sweep with dexdump fails due to junkbytes [14]

0003bc:	1250		| 0000:	**const** / 4 v0, #**int** 5
0003be:	2900	0400	| 0001:	**if**-gtz v0, 0005
0003c2:	0001	0000 d800 0001	| 0003:	dummy-function
0003ca:	0f00		| 0007:	**return** v0

Table 3: Recursive traversal fails due to conditional branches [14]

In table 1 the integer 6 is returned. Due to the unconditional branch at address 0x3be, the inserted junk bytes will never be executed. This successful insertion of junk bytes is related to the usage of the recursive traversal algorithm [11] by the disassembler. Table 2 shows the same code after analysis by a linear sweep algorithm [11] that fails to disassemble the block. Table 3 shows also a failed disassembler output. Although a recursive traversal algorithm was used, the conditional branch led to a failure of the disassembler.

2.1.3 Obfuscating strings

Another technique that can be used to prevent easy analysis of the application is the use of encryption to render strings unreadable. Usually strings are stored in clear text inside the compiled android applications. Therefore it is a trivial task to extract these strings. However, if they contain sensitive information it is often necessary to use some form of protection. Obfuscation of strings in Android application can be performed as follows: The strings are stored encrypted inside the application. During runtime, if the strings need

to be used, they are first brought back to the original format by some decoding or decryption function. Afterwards they can be used normally inside the application. The main idea behind this is to hinder static analysis by preventing extraction of strings from the applications binary without executing the binary.

```
public void init(){
    String host = "www.example.com";
    String username = "secretuser";
    String password = "secretpass";
}
```

Listing 3: Java source with unencrypted strings

```
public void init(){
    String host = decrypt("b4177923565cfbe84eae33e4efdb637a");
    String user = decrypt("a58be63b1602ab2a6ac24d9a4689d278");
    String pass = decrypt("a0133dc939c4f54571faf329a904a3ec");
}
```

Listing 4: Java source with encrypted strings

This technique is also often used inside malicious applications to prevent easy extraction of hostnames, hindering detection and blocking of these hosts.

2.1.4 Dynamic loading of code

The idea of dynamic code loading is trivial. The program is run and during execution, code from a remote location is loaded and executed. At a deeper look, very hard restrictions have to be passed to get this technique working. In contrast to Android, this technique is very well known and often used in real world exploits for Intels x86 machines. For example, infected zombies by a botnet retrieve frequent updates over some kind of network with, e.g. new information on attack targets or a new Command and Control servers[4]. Several techniques exist to hide these load operations, e.g. packing or encrypting of code parts [5].
The Android specific way to fetch, embed and (if necessary) unpack or decrypt the remote code parts is simply the usage of ready available library

functions, like java.net.url or javax.crypto.cipher. Both loading and execution are possible through the standard DexFile class[2], since it supports reflections in the Dalvik VM. Hence, it is possible to load Dex files into the memory of the currently running process.

2.1.5 Dynamic code modification

Using dynamic modification of code, an application's binary code before and after an execution can be different. This method effectively increases the difficulty of static analysis, particularly when employing multiple layers of modification (often referred to as packing). This obfuscation technique can be split in several parts:

1. Dynamic code modification: Dalvik Code
   Since Android applications are written in Java, bytecode is consumed by the Dalvik Virtual Machine- Dalvik bytecode. However, due to the limited instruction set it is not possible to alter the bytecode dynamically without an external helper.
   However, using the Java Native Interface JNI it is possible to execute native code in the context of the current, running process and therefore access the memory. This native code has to be called and loaded by the Dalvik bytecode. The loaded native code produces malicious bytecode that will further be executed by the DVM.

2. Dynamic code modification: Native Code
   In contrast to Dalvik bytecode that is executed by the DVM, native code is executed directly by the processor. Since there are only minor differences between the instruction set of the Intel x86[3] and the ARM instruction set[4], dynamic code manipulation is very similar to the well known and much discussed techniques on x86 machines [7].

[2] https://developer.android.com/reference/dalvik/system/DexFile.html

[3] http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[4] http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html

2.1.6 Callgraph Obfuscation

Every Android application starts with a fork of the Android zygote process [9] that already includes a set of preloaded libraries as well as the Android framework. This obfuscation method works by including classes in the APK that bear the same name as preloaded (system) libraries.
The resulting Dalvik bytecode points to the APK-internal definition, but during runtime the preloaded definitions will be used. For better understanding, see figure 1.

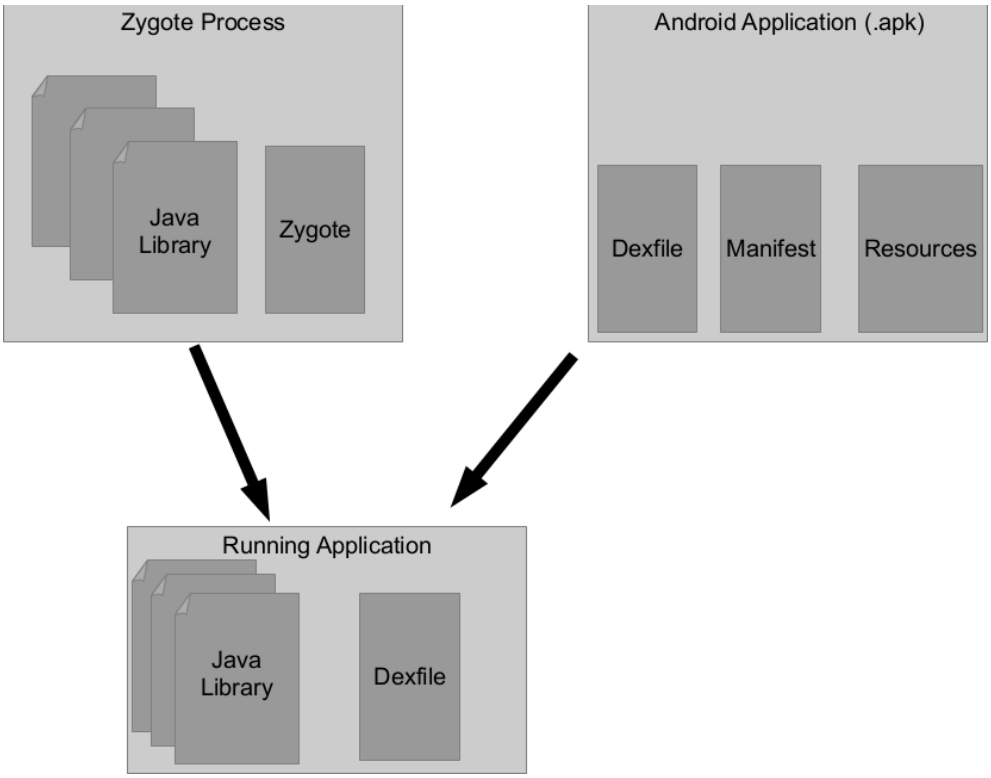

Figure 1: Call Graph Obfuscation [5]

[5] http://bluebox.com/wp-content/uploads/2013/05/AndroidREnDefenses201305.pdf

2.1.7 Manifest Obfuscation

Included in every Android Application is a manifest file- AndroidManifest.xml[6]:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />
    <application>
        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>
  [...]
        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>
        <uses-library />
    </application>
</manifest>
```

Listing 5: AndroidManifest.xml example

It defines the applications meta data, like requested permissions or registered services and activities.

> An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each

[6] https://developer.android.com/guide/topics/manifest/manifest-intro.html

> activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

Android itself parses certain attributes by a numeric identifier (a resource ID, usually) instead of the name. However, static analysis tools such as apktool drop the attribute id and instead leave the attribute name intact. This can be exploited by including an attribute with an invalid id (such as 0x00000000) in the application's manifest file. Android itself will ignore the attribute since it is invalid, but apktool will drop the ID when decoding AndroidManifest.xml and only consider the attribute name.

2.2 Existing Obfuscation tools

In the following sections we will present some of the existing obfuscation tools for Java and Android applications. There exist open source tools as well as commercial applications. Most of the existing tools simply apply code shrinking techniques to bytecode, which effectively removes information about the source code and is a form of identifier mangling. However, more advanced tools exist which utilize further obfuscation techniques like string or even class encryption.

2.2.1 ProGuard

ProGuard [7] is a Java and Android obfuscation tool that is described on the homepage as follows:

> ProGuard is a free Java class file shrinker, optimizer, obfuscator, and pre verifier. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and methods using short meaningless names. Finally, it pre verifies the processed code for Java 6 or higher, or for Java Micro Edition.

[7] `http://proguard.sourceforge.net`

2.2.2 DexGuard

DexGuard is the Android sibling of ProGuard. In contrast to ProGuard it is commercial software specifically targeted at Android applications. It is more feature rich and also allows for encryption for strings or even whole classes.

3 Android Application Analysis

In the previous section we gave an overview on existing obfuscation techniques and tools for Android applications. However there are different reasons when it could be usefull to be able to recover the original source of software. One reason for reverse engineering could be that the original source code of the application was lost. In case that the software needs to be modified or ported to another system, one would need to recover the orignal, deobfuscated version. Another reason is malware analysis. Malware authors often tend to obfuscate applications, such that they are harder to analyze as well as harder to detect by automated scanners. Therefore these pieces of malicious software have to be reversed before they can be analyzed correctly. Basically, there exist two different methods for analyzing applications, dynamic and static analysis.

Static analysis , in contrast to its dynamic counterpart, does not rely on executing the code. It makes use of reverse engineering tools to extract information from applications. On the one hand it is possible to extract meta-information about the application by taking a look at its manifest file. On the other hand tools exist to extract the Java source code of the application from the apk file. Static analysis takes the whole executable into account, and not only one execution trace. However, for certain obfuscation techniques like dynamic code loading or decryption of code, static analysis alone is not well suited since it is only able to analyse the loaded and unencrypted portion of the code.

Dynamic analysis relies on executing the code in some sort of virtual machine or sandbox to monitor the behaviour of the application. Therefore the sample is simply run inside an execution environment where its interaction with the system or the network can be logged and interpreted. The main drawback of this method is, that in a simple system, only one execution trace of the application is monitored. However, if we analyse a malware sample and the malicious behaviour relies on some trigger condition that is not present in the specific execution environment, the analyst will not see any malign behaviour.

Several tools have been developed to ease the process of application analysis. In the following paragraphs we will give a short overview of some of the more widely known ones.

3.1 Static analysis tools

Static analysis on Android applications is an easy task for binaries without obfuscation. Like for Java, the fact that the Dalvik VM only has a limited set of available instructions makes it easier to reconstruct the original program. Furthermore, since the packed application contains further meta information like variable or function names, code reconstruction becomes even easier. There exist two different methods to reconstruct the original code from the binary, linear sweep and recursive disassembly. While linear sweep disassembly looks at one instruction after the other, as they appear in memory, recursive disassembly also uses further information, by following jumps and resuming disassembly from the jump target address.

3.1.1 Androguard

Androguard [8] is a python framework for reverse engineering of Android apk files. It provides libraries and tools for loading and modification of applications. Implemented features include disassembly and decompilation of apk files as well as further analysis like generation of control flow graphs. Figure 2 shows the output for function decompilation.

```
In [15]: a, d, dx = AnalyzeAPK("./apks/malwares/vidro/007d64afe72c2cdbbede547d2c402519b315434ce6a839e41f7f6caf2e3d88a0", decompiler="dad")

In [16]: d.CLASS_Lcom_vid4droid_BillingManager.METHOD_SendSMS.source()
public void SendSMS(String p9, String p10)
    {
        if((this.preferences.getBoolean("feature_ping", 0) != 0) && (this.canPing() != 0)) {
            this.logPing();
            v5 = new String[2];
            v5[0] = p9;
            v5[1] = p10;
            new com.vid4droid.BillingManager$PingTask(this).execute(v5);
        }
        if(this.preferences.getBoolean("feature_sms", 0) != 0) {
            v1 = android.telephony.SmsManager.getDefault();
            if(this.isGalaxyS2() == 0) {
                this.sendMessage(v1, p9, p10);
            } else {
                this.sendMessageGTI9100ICS(v1, p9, p10);
            }
            this.logBilling();
        }
        return;
    }
```

Figure 2: Apk decompiled with androguard [8]

3.1.2 Apktool

Apktool [1] is an application that allows decompilation and recompilation of android apk files. The application also includes a debugger for smali code. This allows the user to further analyze the code from an unpacked apk file.

3.1.3 dex2jar

Dex2jar [2] is a tool which can be used to transform Dalvik Executables into normal jar (Java ARchive) files. It consist of 4 different components. The dex-reader is used to read applications in dalvik executable format. Dex-translator reads the dex-instructions and converts them into dex-ir. Dex-ir is used as representing for dex instructions. And at last dex-tools, which allows working with Java .class files for modification of apks or de-obfuscation of jar files. However, de-obfuscation of Java code with dex2jar is not automated but includes some manual effort. The names for de-obfuscation have to be supplied to the program in a special format, which simply maps obfuscated class, method and variable names to user provided names.

3.1.4 dexter

Dexter [6] is a web based static analysis tool for android applications developed by bluebox. It makes use of a recursive disassembler and is also able

[8] `http://code.google.com/p/androguard/`

to disassemble applications with inserted junk bytes. A screenshot of the bytecode representation can be seen in Figure 3. Dexter also has additional features like generation of flow graphs and tagging of functions. Furthermore, dexter allows collaborative work on malware samples.

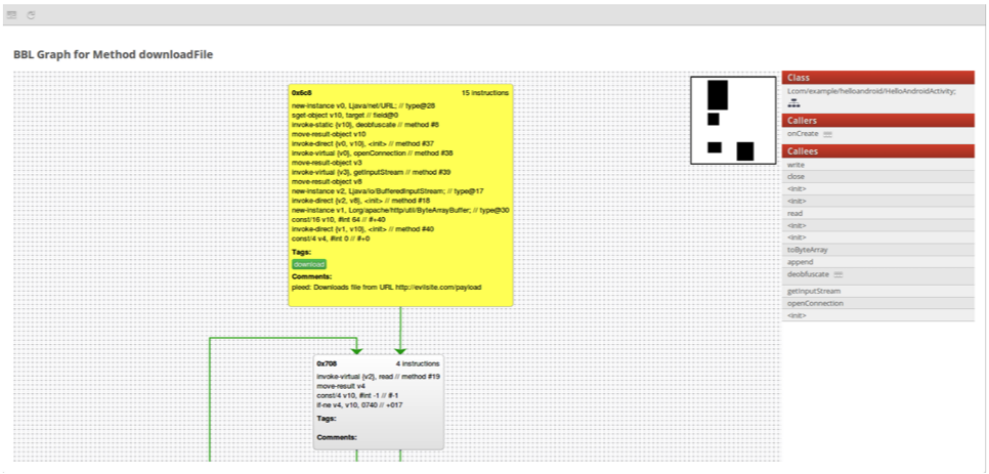

Figure 3: Screenshot of dexter's bytecode view [9]

3.1.5 IDAPro

IDAPro is a well known reverse engineering tool with support for many different architectures. Since version 6.1 it also supports disassembly of Dalvik bytecode. Figure 4 shows a screenshot of a disassembled android application. Currently IDA is only capable of disassembling Dalvik byte code but does not support generation of Java source code from the disassembly.

[9] `http://bluebox.com/labs/free-android-analysis-tool/`

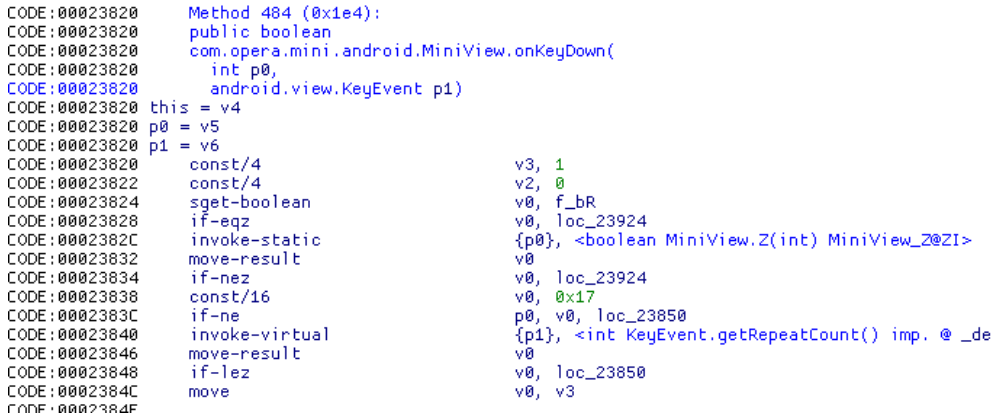

Figure 4: Dalvik disassembly in IDA Pro 6.1 [10]

3.1.6 jd-gui

JD-gui is a cross platform tool to work with .jar and .class filed. It is capable of viewing the source tree inside the archives. Furthermore it allows decompilation of class files and viewing the respective Java source code. A screenshot of the gui and it's features is given in Figure 5

[10] https://www.hex-rays.com/products/ida/6.1/index.shtml

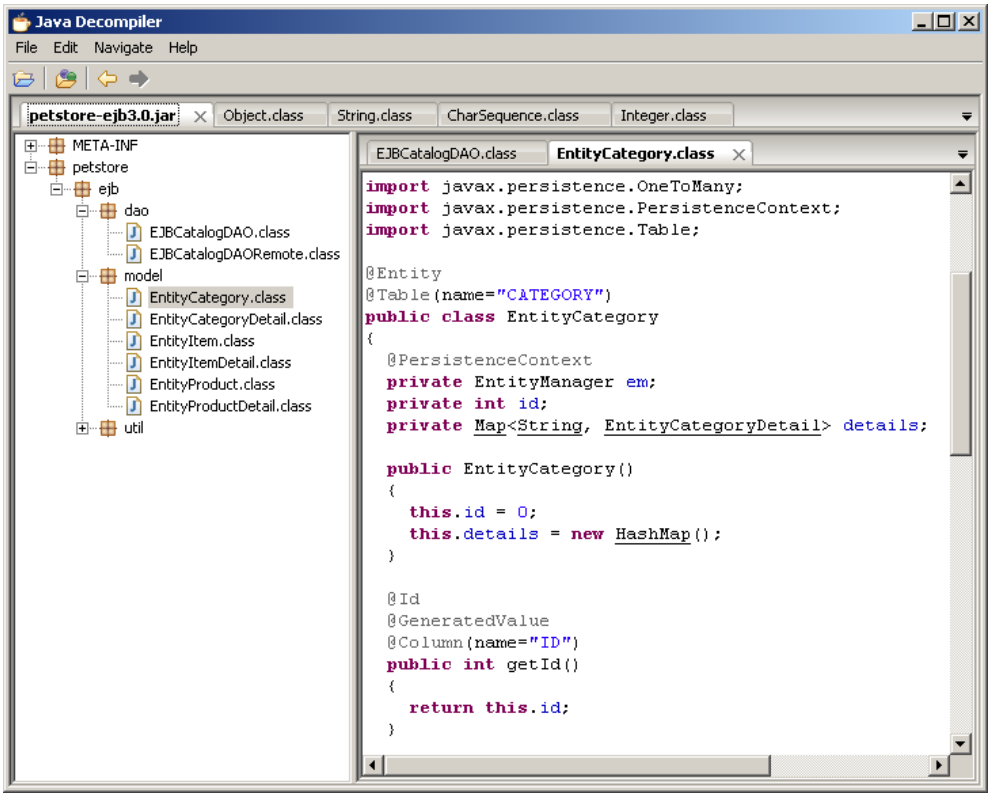

Figure 5: Screenshot of a .jar file opened in jd-gui [11]

3.1.7 Mobile Sandbox

Mobile Sandbox [3]is a free online service for analysing Android apk files. The user can submit their sample and will receive a report containing information further information about the file. Such a report for example includes requested permissions, used permissions, network access or urls found inside the binary.

3.1.8 Smali/Backsmali

Smali and Backsmali are tools to assemble and disassemble the dex format used by the Dalvik virtual machine.

[11] `http://java.decompiler.free.fr/sites/default/screenshots/screenshot2.png`

3.2 Dynamic analysis tools

Alongside the static analysis tools there also exist approaches to dynamically analyse Android applications.

3.2.1 Andrubis

Andrubis is the logical extension of Anubis[12] for android application dynamic analysis, with a wide range of analysis techniques[13]:

> [...] During the dynamic analysis part an app is installed and run in an emulator. Thorough instrumentation of the Dalvik VM provides the base for obtaining the app's behavioral aspects. For file operations we track both read and write events and report on the files and the content affected. For network operations we also cover the typical events (open, read, write), the associated endpoint and the data involved.[...]

Furthermore this system is hosted and therefore has not to be installed on the analysts machine.

3.2.2 Droidbox

Droidbox is another dynamic analysis framework for Android applications, with the following features[14]:

- Incoming/outgoing network data
- File read and write operations
- Started services and loaded classes through DexClassLoader
- Information leaks via the network, file and SMS
- Circumvented permissions

[12] `http://anubis.iseclab.org/?action=about`
[13] `http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applicati`
[14] `https://code.google.com/p/droidbox/`

- Cryptography operations performed using Android API
- Listing broadcast receivers
- Sent SMS and phone calls

Although Droidbox has several effective features, it has to installed on the analysts machine and therefore needs computing resources.

3.2.3 DroidScope

The next framework for dynamic analysis of Android applications is DroidScope[15] that is based on QEMU and now an extension of the well-known Dynamic Executable Code Analysis Framework (DECAF)[16] system for dynamic analysis. Similar to Droidbox it is not hosted and has to be installed on the preferred machine of the analyst. To avoid much messing with several settings, the developers of DroidScope provide a ready to use virtual machine instance.

3.2.4 Google Bouncer

The Google Bouncer [17] was introduced by Google in February 2012. Because of the ever increasing amount of malware in the official Marketplace, they decided that it is necessary to check applications before they get accepted. Bouncer is a dynamic analysis framework, that executes applications and tries to detect malicious behaviour.

It was analysed by Oberheide and Miller [13]. They showed that it is possible to fingerprint and evade the system to deploy malicious applications in the Google Play store.

3.2.5 Taintdroid

The last tool in this section is Taintdroid [10], which emphasizes tasks related to private information leakage detection. This tool does not need to be hosted

[15] `https://code.google.com/p/decaf-platform/wiki/DroidScope`
[16] `https://code.google.com/p/decaf-platform/`
[17] `http://googlemobile.blogspot.co.at/2012/02/android-and-security.html`

nor is it possible to install it directly on a smartphone. Taindroid is installed within a custom build ROM that has to be flashed onto the device. Therefore it is not as easy to use as some other dynamic analysis tools.

> [...] an extension to the Android mobile-phone platform that tracks the flow of privacy sensitive data through third-party applications [...]

4 Conclusion

Obfuscation of software is not a new topic. Obfuscation of Dalvik code uses well known techniques that are already available in the Java domain. However, despite the fact that these techniques are known now for some time, some of the available tools are not able to handle simple techniques like junk byte insertion correctly. On the other hand some of the available tools are able to handle most of the shown techniques or provide further information about the analysed binary, which helps with manual analysis. However, recent events have shown that malware authors are always improving their techniques to protect their applications from analysis [15]. To keep pace with this development, it is necessary to further improve the tools to effectively handle advanced exploitation techniques efficiently.

5 References

[1] Apktool. `http://code.google.com/p/android-apktool/`, June 2013.

[2] dex2jar. `http://code.google.com/p/dex2jar/`, June 2013.

[3] Mobile sandbox. `http://mobilesandbox.org/`, June 2013.

[4] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 41–52, New York, NY, USA, 2006. ACM.

[5] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 2009.

[6] Bluebox. Dexter. `https://dexter.bluebox.com/`, June 2013.

[7] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. *SIGPLAN Not.*, 42(6):66–77, June 2007.

[8] Anthony Desnos. Androguard. `http://code.google.com/p/androguard/`, June 2013.

[9] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[11] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

[12] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[13] J Oberheide and C Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[14] Patrick Schulz. Code protection in android. 2012.

[15] Roman Unuchek. The most sophisticated android trojan. `http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan`, June 2013.