Institute of Computer Science 4
universitätbonn Communication and Distributed Systems

Rheinische
Friedrich-Wilhelms-Universität Bonn,
Germany

Lab Course: Communication and Communicating Devices

Code Protection in Android

Author: Patrick Schulz
Advisor: Daniel Plohmann
Seminar: Communication and Communicating Devices
Semester: Winter term 2011/12
Date: June 7, 2012

Contents

In recent years the Android [1] platform has been become very popular and now runs on more than half of the worldwide sold smartphones [2]. As a result, protecting applications running on Android becomes of interest. Currently, Reverse engineering of Android applications is much easier than on other architectures, due to the high level but simple bytecode language used.
Obfuscation techniques can be used to protect intellectual property of software. This paper discusses possible code obfuscation methods on the Android platform. With a focus on obfuscating Dalvik bytecode, we will also discuss limitations and problems of current reverse engineering tools.

1. Introduction

The Android [1] platform developed by the "Android Open Source Project" is one of the most popular systems for mobile devices in the recent years [2]. This platform is designed in a way that the user can download and install new applications easily. Due to the popularity of the platform, the market for Android applications has grown massively both in variety and financial volume. This results in an increasing interest to protect program code of Android applications. This protection shall help against software piracy and serve as a method to guard intellectual property. On the Android platform there is a further motivation to protect the program code. It is not unlikely for a malware developer to abuse existing applications by injection of malicious functionalities and consequent redistribution of the trojanized versions [3]. Application developers are interested in protecting their applications. Protection in this case means that it should be hard to understand what an application is doing and how its functionalities are implemented.

Obfuscation is the paradigm of hiding program semantics through choosing semantically equivalent but complex and ambiguous representations in order to aggravate analysis. In order to achieve this protection, obfuscation transforms program code of an application in a way that it is "hard to reverse engineer" but without changing the behavior of this application. Hard to reverse engineer means that automated programs cannot produce good abstractions of the analyzed program and the results of the analysis become harder to understand for a human analyst.

Reverse Engineering on the other hand has the goal to gather as much information on an application as possible in order to understand its functionalities. The ideal result of a reverse engineering process for an Android applications would be to reconstruct the original Java source code out of the distributed binary form. Obfuscation cannot prevent reverse engineering but can make it harder and more time consuming. We will discuss which obfuscation and code protection methods are applicable under Android and show limitations of current reverse engineering tools.

This paper is organized as follows. In section 2 we discuss related tools, which can be used to obfuscate and reverse engineer an Android application. Section 3 describes the Android architecture and how an application is structured. Obfuscation techniques are discussed and evaluated in section 4. In the final section we conclude and summarize our findings.

2. Related Work

Code obfuscation is a well known method and extensively researched on other architectures such as x86. It is used to protect intellectual property of software and is also widely used in malware both to evade detection by Anti-Virus products and harden the code against analysis. It can have a huge impact on the effectiveness of reverse-engineering, e.g. by disabling the usefulness of tools and circumventing heuristics. In the following section we will discuss related tools for Android code obfuscation as well as reverse engineering tools.

2.1. Code protectors

The following tools can be used within the deployment process of an application to obfuscate program code and protect the application against analysis.

ProGuard

ProGuard [4] is an open source tool which is also integrated in the Android SDK [5]. It can be easily used within the development process. ProGuard is basically a Java obfuscator but can also be used for Android applications because they are usually written in Java. The feature set includes identifier obfuscation for packages, classes, methods, and fields. Besides these protection mechanisms it can also identify and highlight dead code so it can be removed in a second, manual step. Unused classes can be removed automatically by ProGuard.

Without proper naming of classes and methods it is much harder to reverse engineer an application, because in most cases the identifier enables an analyst to directly guess the purpose of the particular part. The program code itself will not be changed heavily, so the obfuscation by this tool is very limited.

Allatori

Allatori [6] is a commercial product from Smardec. Besides the same obfuscation techniques like ProGuard, shown in section 2.1, Allatori also provides methods to modify the program code. Loop constructions are dissected in a way that reverse engineering tools cannot recognize them. This is an approach to make algorithms less readable and add length to otherwise compact code fragments. Additionally, strings are obfuscated and decoded at runtime. This includes messages and names that are normally human readable and would give good suggestions to analysts.

The obfuscation methods used in Allatori are a superset of ProGuards so it is more powerful but does not prevent an analyst from disassembling an Android application.

2.2. Analysis tools

The usual reverse engineering process makes use of a whole range of different analysis methodologies and tools. In this case we only consider static analysis tools. In the follow-

ing, we present a selection of tools that can be used to disassemble Dalvik bytecode, see section 3.2. Furthermore, meta information, e.g. method identifier and string constants, about the program structure can be gathered, which helps to identify interesting parts of an application.

dexdump

"Dexdump" is a tool that is directly included with the Android SDK [5]. It is a basic dex file dissector and can also disassemble Dalvik bytecode, which is stored in the dex file. dexdump uses a trivial and straight forward approach in order to disassemble a dex file. It uses linear sweep to find instructions, this means that dexdump expects a new valid instruction behind the last byte of the currently analyzed instruction. This is true in most benign cases, but must not be true in case of obfuscated bytecode, as shown e.g. in section 4.4 on Junkbytes. Furthermore, dexdump outputs the embedded classes, methods and fields as well as some detailed information about the class structure.

smali

The smali [7] program is an assembler, which is the opposite of the already explained disassembler. smali brings its own disassembler called "baksmali". So in combination, both tools can be used to unpack, modify, and repack Android applications. The interesting part for obfuscation and reverse engineering is baksmali. baksmali is similar to dexdump but uses a recursive traversal approach to find instructions. So in this approach the next instruction will be expected at the address where the current instruction can jump to, e.g. for the "goto" instruction. This minimizes some problems connected to the linear sweep approach. baksmali is also used by other reverse engineering tools as a basic disassembler.

Androguard

A very powerful analysis tool is Androguard [8]. It includes a disassembler and other analysis methods to gather information about a program. Androguard helps an analyst to get a good overview by providing call graphs and an interactive interface. The integrated disassembler also uses the recursive traversal approach for finding instructions like baksmali, see section 2.2. At the moment, Androguard is one of the most popular analysis toolkits for Android applications due to its big code base and offered analysis methods. It is also used as backend component for other tools like apkinspector [9].

IDA Pro

IDA Pro [10] is well known as a powerful tool for reverse engineering under x86. It also supports many other architectures as well as Dalvik bytecode. IDA Pro provides a graphical interface and supports plug-ins in order to extend analysis functionality. Besides the usage of the recursive traversal approach, IDA Pro can start disassembling at specific points within the file by a user request. This is useful in the case when the disassembling algorithm missed some instructions. A very helpful feature of IDA Pro is the presentation of the Dalvik code as a graph. This makes it much easier for an analyst to follow the control flow within a program.

3. Android Architecture

In this section we describe the main parts of the Android architecture. This includes the runtime environment and the format of Android applications as well as the build process. These parts can be affected by obfuscation methods as described in section 4. The reverse engineering process and its components are discussed in the last part of this section.

3.1. Android applications and runtime environment

Android applications are written in the Java [11] programming language and deployed as files with an ".apk" suffix, later called APK. It is basically a ZIP-compressed file and contains resources of the application like pictures and layouts as well as a dex file. This dex file, saved as "classes.dex", contains the program code in form of Dalvik bytecode. Further explanations on this bytecode format are given in section 3.2. The content of the APK is also cryptographically signed, which yields no security improvement but helps to distinguish and confirm authenticity of different developers of Android applications.

Applications can be downloaded to an Android device via USB cable, from the official Android Market [12], or various third party markets on the Internet. The installation process on the device will be controlled by the Android platform, which also takes care about permission assignment. Within the installation process, every installed application gets its own unique user ID (UID) by default. This means that every application will be executed as a separate system user.
When an Android application is executed, the process consists of the following four parts:

- Dalvik bytecode, which is located in the dex file
- Dalvik Virtual Machine [13], which executes the Dalvik bytecode
- Native Code, like shared objects, which is executed by the processor
- Android Application Framework, which provides services for the application

The Dalvik Virtual Machine (DVM) provides also the ability to call native functions within shared objects out of the Dalvik bytecode. When speaking of reverse engineering an Android application we mostly mean to reverse engineer the bytecode located in the dex file of this application.

3.2. Dalvik bytecode

The program code of an Android application is delivered in form of Dalvik bytecode. It will be executed by the Dalvik Virtual Machine and is comparable to Java bytecode. The main difference between the Java Virtual Machine (JVM) and the DVM is, that JVM is a stack based machine whereas DVM is register based. The Dalvik bytecode format [14] has been developed for Android to be more efficient and smaller due to the limited resources on mobile devices. Dalvik bytecode of an application is normally not optimized, because it is executed by a DVM which can run under different architectures. So there is a separate optimizing step while installing an application, where the bytecode gets optimized for the underlaying architecture. The optimized form is also called "odex". The optimization is

done by a program called "dexopt" which is part of the Android platform. The DVM can execute optimized as well as not optimized Dalvik bytecode.

3.3. Build process of an Android application

An Android application needs many steps and tools until the APK is build and ready to be deployed. As mentioned in section 3.1, Android applications are written in the Java programming language. Therefore, the application code is available in ".class" files. This plain text files are provided by the application developer. The first step of the build process, as shown in figure 1 step 1, starts with the Java files which will be compiled into ".class" files by a Java Compiler. This step is similar to a Java program build process. At this point the class files contain Java bytecode representing the compiled application. An optional step "a" is to apply a Java Obfuscator [4][6] on this ".class" files. The next step is the transformation from Java bytecode into Dalvik bytecode, as discussed in section 3.2. For this, the "dx" program is used in step 2. It is included in the Android SDK [5] due to it is necessity for building an application for the Android platform. The output of dx will be saved into a single dex file "classes.dex". This file will be included in an APK in a later step. Before this happens, it is possible to apply a further obfuscator operating on Dalvik bytecode, as shown in figure 1 step b. Such Dalvik obfuscation techniques will be discussed in section 4. The last step of the build process is packing and signing the APK. The ApkBuilder constructs an apk file out of the "classes.dex" file and adds further resources like images and ".so" files. ".so" files are shared objects which contains native functions that can be called from within the DVM. The "jarsigner" just adds the developer signature to the APK, which can now be installed on an Android device. We should further mention that the obfuscator ProGuard [4] works at position "a", has been integrated in the default development process and is recommended to use for release versions.

3.4. Reverse engineering

Reverse engineering is the process of gaining information about a program and its implementation details. This process aims at enabling an analyst to understand the concrete relation between implementation and functionality of the program. An optimal output of such a process would be the original source code of the application, but this is not feasible in general. An understanding of the program can be reached without recovering a form that recompiles to the original program. Therefore, it is necessary for such a process to provide on the one hand abstract information about structure and inter-dependencies and on the other hand result in very detailed information like bytecode and mnemonics that allow interpretation of implementation. The produced abstract representation of the program code, e.g. class diagrams, helps an analyst to get a quick overview and can give good suggestions where to start with further investigations. This means we use abstract representations to find starting points and then use more detailed ones to analyze these interesting parts. It is also likely to generate intermediate representations e.g. an intermediate language which is more abstract than mnemonics in order to enable an analyst to read the program code faster. In this paper we focus on the disassembling process which yields mnemonics because this is the fundamental step in reverse engineer-

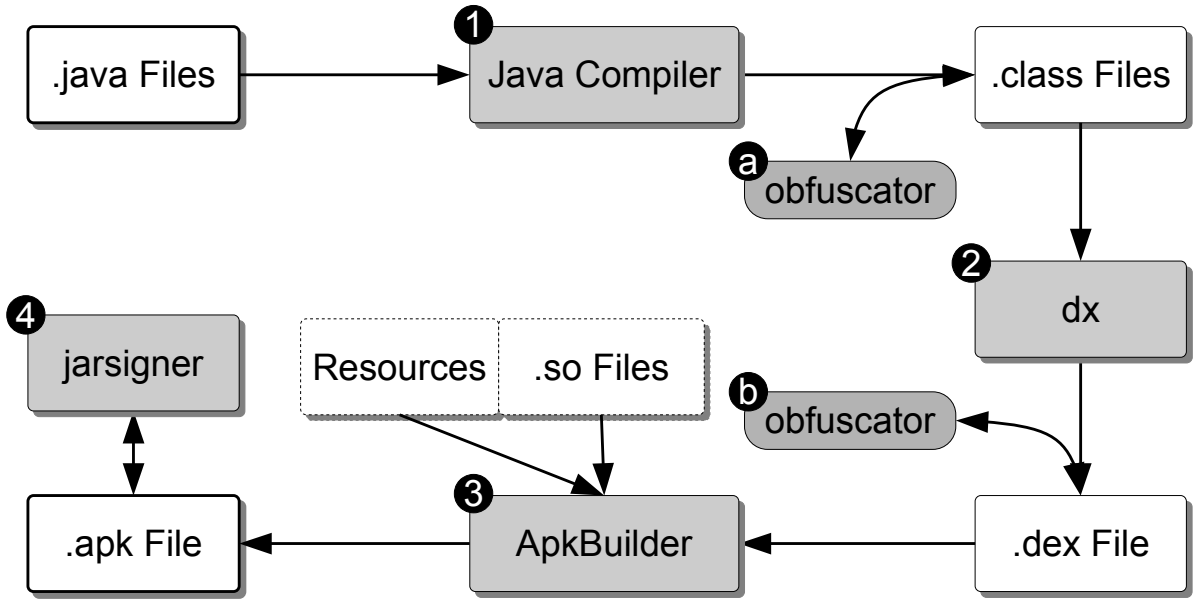

Figure 1: Android application build process [15].

ing due to the fact that most of the other processes are based on the disassembling process.

A disassembler is a tool which gets bytecode as input, e.g. the binary executable, and produces the equivalent mnemonics. Some examples of disassemblers have already been introduced in section 2. We will come back to them in the obfuscation process in section 4. Mnemonics represent bytecode in a human readable form and are the output of the disassembling process. This representation yields therefore nearly the same information as its bytecode counterpart. Only a few pieces of information are usually not included like the addresses of instructions. In figure 2, the output of the disassembler "dexdump"[5]

```
1 <Dalvik bytecode>          | <Mnemonics>
2 1250                       |const/4 v0, #int 5 // #5
3 1201                       |const/4 v1, #int 0 // #0
4 3d01 0400                  |if-lez v1, 0006 // +0004
5 d801 0101                  |add-int/lit8 v1, v1, #int 1 // #01
6 d800 0009                  |add-int/lit8 v0, v0, #int 9 // #09
7 0f00                       |return v0
```

Figure 2: Disassembled method of an Android application generated by dexdump[5].

to a short method is shown. On the left side the binary representation called bytecode is listed. They are printed as hexadecimal values. Each line represents exactly one instruction of the DVM. Due to unintuitive representation, the right side also shows the corresponding mnemonics. The meaning of every instruction within the bytecode is well defined [14] but the syntax of the mnemonics can be chosen by disassemblers freely. At the moment, there are two mainly used syntaxes [14][16]. The transformation from bytecode

to mnemonics and vice versa is available due to the bijective mapping, but finding the correct start address and offset can be challenging. There are two major approaches: linear sweep disassembling and recursive traversal disassembling [17]. The linear sweep algorithm is prone to producing wrong mnemonics e.g. when a assembler inlines data so that instructions and data are interleaved. The recursive traversal algorithm is not prone to this but can be attacked by obfuscation techniques like junkbyte insertion as discussed in section 4.4. So for obfuscation, a valuable attack vector on disassembling is to attack the address finding step of these algorithms.

4. Obfuscation techniques

Obfuscation techniques are used to protect software and the implemented algorithms. They are well known under the x86 architecture for years, where it is still an extensively researched topic. These techniques are designed to make reverse engineering harder and more time consuming, as discussed in section 3.4. On the other hand also new reverse engineering techniques were developed and improved. So, it is an ongoing development of both sides.
In general, the applicaiton of obfuscation techniques must not alter the behavior of programs, but their outer representation. Obfuscation techniques often only target specific reverse engineering steps due to the circumstance that there are few general protection schemes. Depending on the concrete obfuscation technique, a possible drawback is an impact on execution speed. In this paper we will not discuss such slow downs in detail because the presented techniques do not cause substantial slow downs.
In the following list we define reverse engineering goals, which should be covered and attacked by obfuscation techniques.

- Derivation of correct mnemonics according to the bytecode, so that they are consistent with the executed instructions.
- Extraction of meta information like identifiers and strings.

4.1. Identifier mangling

Identifiers are names for packages, classes, methods, and fields. They are represented as strings. In figure 3, a snippet of Java source code with highlighted identifiers is shown.

```
1 public class NetworkManager {
2  private String encrypt( String input )
3  { ... }
4  public void send( String input )
5  { ... }
6 }
```

Figure 3: Java source code example with highlighted identifiers.

In the example it is easy to get an idea of what this class is about and that it is almost

```
1 public class a {
2  private String a( String ab )
3  { ... }
4  public void b( String ac )
5  { ... }
6 }
```

Figure 4: Java source code example with obfuscated identifiers.

certain that some kind of encryption is involved. From the example it is obvious, that original identifiers give information about interesting parts of a program. Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed. Identifier mangling aims to neutralizing these information in order to prevent this reduction.

Identifier mangling tries to remove any meta information about the behavior, which can be gathered out of identifiers. An easy approach is to replace any identifier with a meaningless string representation holdin respect to consistence. This means identifiers for the same object must be replaced by the same string. As for the substitution schema, random strings do not contain any information about the object or its behavior itself, so they fulfill the requirement.
The obfuscator ProGuard, presented in section 2, uses a similar approach. It uses minimal lexical-sorted strings like {a, b, c, ..., aa, ab} instead of random strings, which is shown in figure 4. This has the advantage of minimizing the memory usage, which should also be concerned in the context of mobile devices. Such an obfuscator can be applied within the development process in step "a" or step "b" in figure 1.

4.2. String obfuscation

Strings are arrays of characters which are frequently used within a program e.g. for enabling user interaction or printing messages. In comparison to identifiers, see section 4.1, the original content of a string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog. In figure 5 an example of string usage is shown.

```
1 private void start() {
2  String server = "irc.cnc.mal";
3  String password = "notpublic";
4  connection.connect( server, password );
5 }
```

Figure 5: Java source code example with highlighted strings.

A reverse engineering process can extract at least two important information out of such strings. On the one hand we can identify functions of interest by interpreting the context in which strings are used and on the other hand we get the content of these strings, e.g.

server addresses, passwords, or cryptographic keys. Therefore, an Android application developer is interested in obfuscating these strings.
String obfuscation can be achieved by any injective function $\mathcal{F}$ which is invertible and transforms an arbitrary string into another string. We can use e.g. the xor function or encrypt a string using AES [18]. In order to obfuscate a string $\mathcal{S}$, we replace $\mathcal{S}$ within the program code with the output of $\mathcal{F}(\mathcal{S})$. Beside the output of $\mathcal{F}(\mathcal{S})$ the obfuscator has to generate a deobfuscation stub $\mathcal{F}^{-1}$ which has to be included into the program code. This deobfuscation stub is used to reconstruct the original string just before it is used at runtime, which is necessary in order to not change the behavior of the program.
This obfuscation technique does not make it harder to understand the program code due to the fact that it does not change it, besides injecting a deobfuscation stub. This technique aims at reducing the amount of extractable meta information. Furthermore, we have to mention that this obfuscation can be defeated by dynamic analysis in which the program will be executed. Here, an analyst may execute the program until the first time the particular string is used and can then extract the content of this string. The string holds the original value at this point due to the assumption that the behavior is not changed by the obfuscator.

4.3. Dynamic code loading

An optimal obfuscator would transform an application in a way so that it does not contain any meta information or directly interpretable bytecode. This is not possible because in this case also the DVM could not execute this program and it would not be runnable. A result close to this ideal transofmration can be achieved by applying packing that goes beyond obfuscation. This approach is well known under the x86 architecture and is often used by malware [19]. A packer takes an arbitrary executable and transforms it in a way that an analyst cannot immediately extract information out of it. For example, this can be achieved by not only encrypting data but also code, which is then decrypted during runtime before it is executed. In this case, no information can be extracted without decrypting it first. In order to yield an executable application, the packer has to generate a so called unpacking stub, which will be executed when starting the application. This unpacking stub will decrypt the encrypted application, load it into the process context and execute it. In this approach the analyst cannot gather useful information out of the encrypted application. He has to investigate the decryption stub in order to decrypt the application by hand or use dynamic analysis techniques.
The obfuscator has to generate two components, the encrypted application and a decrypter stub. In Android, the encrypted application would be an encrypted dex file. This is rather easy to generate compared to the decrypter stub. The decrypter stub has to implement four main functionalities as shown in figure 6. At first the encrypted application has to be fetched into memory fetched. This can be done by downloading a dex file from a remote server or extracting it from an internal data structure. In Android we can simple add arbitrary files to an APK and access them at runtime. The second step in figure 6 is the decryption of the encrypted dex file, yielding the original dex file. The cryptographic function can be chosen freely, due to its application is completely independent from the content of the dex file. After the unpacking stub has generated the original dex file, it can be loaded into the DVM and executed, as shown in steps 3 and 4 in figure 6.

In order to attack such a schema, a reverse engineering tool has to gain access to the decrypted dex file. This can only be done after step 2 in figure 6.

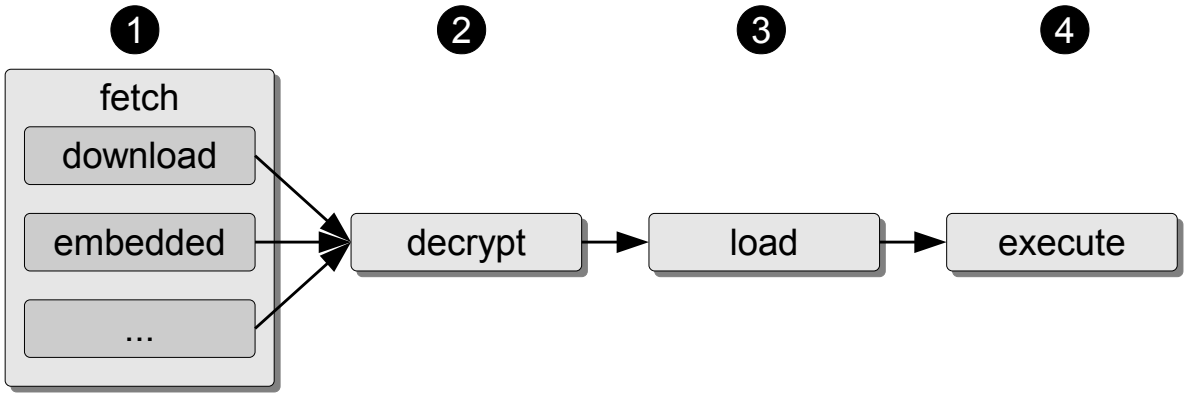

Figure 6: The steps of an decryption stub in case of a packed application.

The implementation of step 1 and 2 in Android is very simple, because we can use existing library functions [20][21]. The 3rd and 4th step are also possible because of the support of reflection in DVM via the "DexFile" class [22]. This class supports loading of dex files into the currently running process, a requirement for step 3 in figure 6. The publicly accessible functions can only operate on dex files stored on the file system, which means that our decrypted dex file has to be saved to the file system before loading. In this case an analyst can easily copy it from there and the considered protection scheme becomes useless. Another problem with this mode of operation is the transformation of the loaded dex file into an optimized equivalent that in turn is stored on the file system. The optimized dex file is secured by file system permissions so that our application cannot delete them to prevent leakage of the decrypted file. From this, we conclude that using these functions are not suitable to create a protection scheme comparable to the likes of x86. However, the "DexFile" class provides another function, which supports loading a dex file from an arbitrary bytearray instead of a file, but this is a private function as shown in figure 6.1a and it cannot be called directly by our application.
To circumvent this restriction we can use the provided "Java Native Interface" (JNI) of the DVM, see figure 7. JNI is intended to allow execution of native code, which is located in shared objects, out of the DVM. This is useful e.g. for computationally complex algorithms like graphic processing. The DexFile class also uses this native interface as shown in step 2a.
While the "DexFile" class does not provide access to all functions, it allows access to those which store optimized dex files. We implemented our own minimal dex file loader as a shared object, which provides the function "public static int openDexFile(byte[] fileContents)", as shown in step 1b and 2b in figure 7. We use the circumstance that the shared object "libdvm.so" exports symbols, which we can use to call this particular function in order to circumvent the restrictions that occur when directly interfacing with the Android library within the DVM.
This approach has two advantages. We do not have to store the decrypted dex file on the file system, because we can pass the content of it via a bytearray. By this, the content

of our dex file is only available within volatile memory. The other advantage is that this function does not generate an optimized dex file. An example implementation can be found in the appendix A.

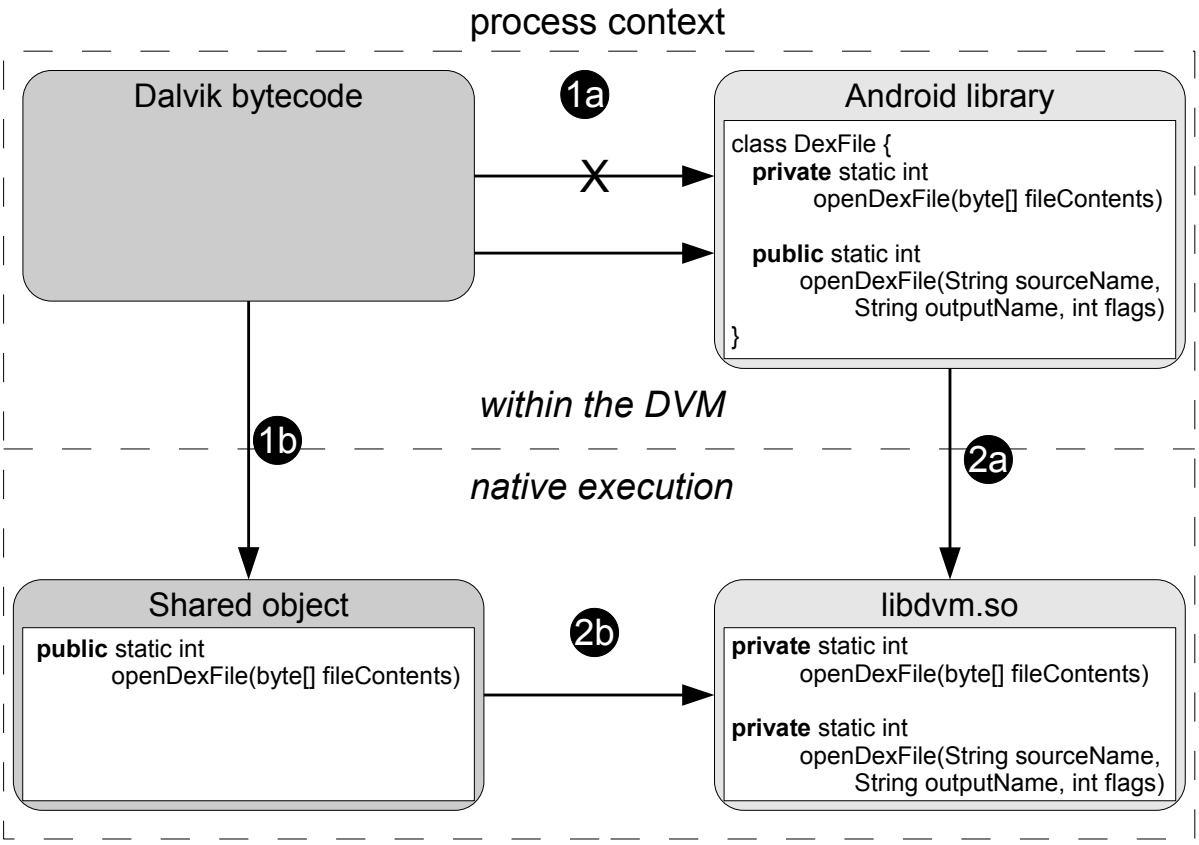

Figure 7: Possible control flows to load a further dex file into a running process.

This protection scheme makes it hard to analyze the target application, because its bytecode is only available encrypted. To get the decrypted version, the unpacking stub has to be analyzed. This slows down the whole reverse engineering process. Another advantage of this approach is that we can apply further obfuscation techniques, but just need to apply them on the unpacking stub. We also can freely modify the unpacker stub in order to fulfill requirements of some advanced obfuscation techniques.

4.4. Junkbytes

Junkbyte insertion is a well known technique under the x86 architecture [23]. It is used to confuse disassemblers in a way that they produce disassembly errors and disallow correct interpretations by an analyst. This is done by inserting junkbytes in selected locations within the bytecode where a disassembler expects an instruction. The position of a junkbyte must take the disassembling strategy into account in order to reach a maximum of obfuscated code. The two major disassembling strategies are discussed in chapter 3.4. Another condition for the location is that the junkbyte must not be executed, because an execution would result in an undesired behavior of the application. So a junkbyte must be located in a basic block which will never be executed. We can ensure that this basic

block will not be executed by adding an unconditional branch in front of it. It is also possible to use a conditional branch while ensuring that the evaluated argument results in a constant value. This technique is called usage of an opaque predicate.

```
0003bc: 1250                    |0000: const/4 v0, #int 5 // #5
0003be: 2900 0400               |0001: goto/16 0005 // +0004
0003c2: 0001                    |0003: <Junkbytes>
0003c4: 0000                    |0004: <Junkbytes>
0003c6: d800 000                |0005: add-int/lit8 v0, v0, #int 1 // #01
0003ca: 0f00                    |0007: return v0
```

Figure 8: Disassembly output with correct detection of the junkbytes.

```
0003bc: 1250                    |0000: const/4 v0, #int 5 // #5
0003be: 2900 0400               |0001: goto/16 0005 // +0004
0003c2: 0001 0000 d800 0001     |0003: packed-switch-data (4 units)
0003ca: 0f00                    |0007: return v0
```

Figure 9: The linear sweep algorithm used by dexdump[5] produces disassembly errors due to junkbyte insertion.

```
0003bc: 1250                    |0000: const/4 v0, #int 5 // #5
0003be: 3c00 0400               |0001: if-gtz v0, 0005 // +0004
0003c2: 0001 0000 d800 0001     |0003: packed-switch-data (4 units)
0003ca: 0f00                    |0007: return v0
```

Figure 10: The recursive traversal algorithm used by most of the disassemblers produce disassembly errors due to junkbyte insertion using conditional branches.

In figure 8, a method is shown which returns 6 as an integer. We have inserted an unconditional branch at the address 0x3BE followed by two junkbytes. The unconditional branch ensures that these junkbytes will not be executed and jumps directly to the addition encoded at the address 0x3C6. This is the correct output that will be produced by a recursive traversal disassembling algorithm. Disassemblers like dexdump, which use the linear sweep algorithm, will produce incorrect output, as shown in figure 9. In this case the "add-int" instruction at address 0x3C6 is omitted.
With this simple junkbyte insertion using an unconditional branch, we can force a linear sweep algorithm to misinterpret the bytecode. In order to also cover recursive traversal algorithms we have to incorporate conditional branches. A simple example is shown in figure 10. This will cover both recursive traversal as well as linear sweep algorithms. In this case the addition is also not visible like in figure 9.
The number of obfuscateable instructions depends on the junkbyte e.g. if the junkbyte indicates the begin of an instruction consisting of multiple bytes, more of the following instructions are covered, which is shown in figure 11 in the third column. So the "packed-switch-data" instruction we used in the examples is a good choice due to its variable length, but not every disassembler processes this instruction because it is a very

junkbyte sequence	reverse engineering tool	covered bytes	output/behavior
0x0001	dexdump, baksmali, dex2jar	6-4kk	overlapping instructions will not be discovered
0x0001	androguard, dedexer	-	no obfuscation due to this instruction is not implemented in this tools
0x18	dexdump, baksmali, androguard, dedexer, dex2jar	9	overlapping instructions will not be discovered
0x3c (*)	dexdump	≥ 0	all instruction following this junkbyte in the particular method are discarded
0x3c (*)	baksmali	≥ 0	baksmali crashes while parsing this junkbyte. No further output
0x3c (*)	androguard	-	ignores unknown opcodes.
0x3c (*)	dedexer	≥ 0	dedexer crashes while parsing this junkbyte. No further output
0x3c (*)	dex2jar	≥ 0	dex2jar crashes. Produces no output.

(*) an unused opcode. Any other unused opcode should produce the same results.

Figure 11: The behavior of different reverse engineering tools when parsing inserted junkbytes.

special one. The "packed-switch-data" instruction is a dummy instruction which cannot be executed, but just stores data. So we evaluated different candidate bytes for junkbyte insertion and compared them according to their impact on different disassemblers. The results are shown in figure 11.

The results show that all tested reverse engineering tools have problems with junkbyte insertion and can be tricked. Besides this some of them crash and have flaws in exception handling. This shows that the disassembler implementations have to be improved and that we still need better algorithms for disassembling in order to find all instructions and to determine the control flow within bytecode correctly. This still holds for a simple architecture like Dalvik, which e.g. does not support indirect jumps. An indirect jump allows to jump according to the content of a register and this makes it generally very difficult to determine the branch address with static analysis in general.
Junkbytes are still effective and can be used to hide instructions within the disassembling process. An analyst has to check this manually, which is time consuming and as a result junkbyte insertion is a valuable technique for obfuscation.

4.5. Self modifying code

Program code which can alter itself at runtime is called self modifying code. This means that the instructions we observe in an initial piece of code, do not need to be the same code which are executed when running the program. This technique can be used to hide "true" code during static analysis, as discussed in section 4.3. It is even possible without

loading further code into the runtime environment.

On the Android platform there are situations where it is possible to have self modifying code. In the following sections we discuss different situations and techniques to enable self modification.

4.5.1. Self modifying Dalvik code

Android applications are mostly written in Java and so they consist of Dalvik bytecode, as explained in section 3.2. Due to the limited instruction set of the DVM compared to the x86 architecture or the ARM architecture, there is no direct way to access the bytecode with an instruction. As a consequence, it is not directly possible to read nor write into the bytecode stream which is executed. Therefore, self modifying code is not possible within the DVM itself.
To achieve modification within a Dalvik bytecode stream we have to use external components. As a result, we can make the assumption that the analyzed code will not be altered, in case of absence of further external component which may implement such modification techniques.

4.5.2. Modifying Dalvik code using JNI

In Order to achieve modification of Dalvik bytecode during runtime, we have to use native code due to the missing ability of self modifying Dalvik bytecode, as discussed in section 4.5.1. This can be done using JNI, which we have used in section 4.3 on dynamically loading code from memory. This interface enables us to execute native code within our current process context. This means that it is possible to access the process memory where the DVM as well as the running Dalvik bytecode are located. The advantage of native code in this case is that we can access arbitrary memory locations, which on the other hand is not possible using Dalvik bytecode.
An example is shown in appendix B. The Java source code and therefore also the Dalvik bytecode consists of three parts:

- A JNI call in order to execute our native code,
- A magic byte constant, which is used to locate our Dalvik bytecode,
- The Dalvik bytecode which should be modified.

The native code will be called by our Dalvik bytecode. It has a search routine that starts to look for the magic byte constant within the memory. This process with the intention to gather position information is well-known from shellcode and called "egg-hunting". After the magic byte constant has been identified, we can modify the target Dalvik bytecode, which is located at a previously measured offset to the magic byte constant. By this we can modify our bytecode. After returning from the native code, the modified bytecode will be executed.
We showed that static analysis of the Dalvik bytecode is not enough in order to reverse engineer an Android application due to the fact that the bytecode can be altered during runtime. A further implication of this is, that an analysis must also cover the native code and cannot only take Dalvik bytecode into account as an isolated part of the application.

4.5.3. Self modifying native code

Native code is code which will be executed by the processor directly. Due to the fact that most mobile devices are based on an ARM architecture, we speak about ARM native code in this section. The ARM instruction set is comparable to the instruction set of the x86 architecture. Due to the fact that self modifying code is well known since years under x86 [24], it is rather easy to transfer these techniques to ARM. A simple example is to map a new memory section into the process, using the syscall "mmap". Now, native code can be written to this new section and be executed by jumping into it. Due to the fact that we can write and execute the code within the section at the same time, we are also able to implement memory-based self modifying code. For this approach we have to take care about the caching behavior of the ARM architecture [25].
ARM native code has no further restrictions on self modifying code in comparison to x86. So we can apply any known obfuscation and even more generally, any protection technique based on self modifying code on Android applications as well. To counter this situation, we can also use the developed analysis techniques from x86 like dynamic analysis.

5. Conclusion

The Android platform is an interesting field of research and also a valuable market for companies. Because source code can be easier recovered from an application in comparison to x86, there is a strong need for code protection and adoption of existing reverse engineering methods. Main parts of Android application functionalities are realized in Dalvik bytecode. So Dalvik bytecode is of main interest for this topic. The Dalvik instruction set is less powerful when compared to e.g. those of the x86 architecture, due to the lack of ability to access the own bytecode and perform indirect jumps. So it should be less complex to work with Dalvik bytecode in order to parse, evaluate and disassemble it. In fact, today's disassemblers have still problems with transforming Dalvik bytecode into mnemonics correctly. Also, the Android system does not prevent modification of this bytecode during runtime. This ability of modifying the code can be used to construct powerful code protection schemata and so make it hard to analyze a given application.
Beside modification of bytecode during runtime other obfuscation techniques can be used in order to protect Android applications. These techniques do not try to make it harder to parse the code but reduce meta information within the applications. This is done to prevent, that information about the application and its functionalities are gathered by an analyst. Those techniques like identifier mangling and string obfuscation are very effective and let the reverse engineering process be time consuming.

In this paper we showed that the existing disassembling implementation have to be improved and that well-known obfuscation techniques as proven on other processor architectures are still effective.

References

[1] Android Open Source Project. Android sources. Visited: May, 2012. [Online]. Available: http://source.android.com

[2] Gartner. Worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. Visited: May, 2012. [Online]. Available: http://www.gartner.com/it/page.jsp?id=1924314

[3] C. A. Castillo and Mobile Security Working Group McAfee, "Android malware past, present, and future," 2011.

[4] E. Lafortune. Proguard. Visited: May, 2012. [Online]. Available: http://proguard.sourceforge.net/

[5] Android Open Source Project. Android sdk. Visited: May, 2012. [Online]. Available: http://developer.android.com/sdk/index.html

[6] Allatori. Allatori obfuscator. Visited: May, 2012. [Online]. Available: http://www.allatori.com/doc.html

[7] smali. Visited: May, 2012. [Online]. Available: http://code.google.com/p/smali/

[8] A. Desnos. Androguard. Visited: May, 2012. [Online]. Available: http://code.google.com/p/androguard/

[9] Apkinspector. Visited: May, 2012. [Online]. Available: http://code.google.com/p/apkinspector/

[10] H.-R. SA. Ida pro. Visited: May, 2012. [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml

[11] Oracle. Java. Visited: May, 2012. [Online]. Available: http://www.java.com

[12] Google Inc. Android market. Visited: May, 2012. [Online]. Available: https://play.google.com

[13] Android Open Source Project. Dalvik and dalvik virtual machine. Visited: May, 2012. [Online]. Available: http://code.google.com/p/dalvik/

[14] Android Open Source Project. Bytecode for the dalvik vm. Visited: May, 2012. [Online]. Available: http://source.android.com/tech/dalvik/dalvik-bytecode.html

[15] Android Open Source Project . A detailed look at the build process. Visited: May, 2012. [Online]. Available: http://developer.android.com/guide/developing/building/index.html

[16] J. Meyer and D. Reynaud. Jasmin. Visited: May, 2012. [Online]. Available: http://jasmin.sourceforge.net/

[17] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," 2002.

[18] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES) (FIPS PUB 197)," 2001.

[19] F. Guo, P. Ferrie, and T.-c. Chiueh, "A study of the packer problem and its solutions," 2008.

[20] Android Open Source Project. Android reference - javax.crypto.cipher. Visited: May, 2012. [Online]. Available: http://developer.android.com/reference/javax/crypto/Cipher.html

[21] Android Open Source Project . Android reference - java.net.url. Visited: May, 2012. [Online]. Available: http://developer.android.com/reference/java/net/URL.html

[22] Android Open Source Project. Android reference - dalvik.system.dexfile. Visited: May, 2012. [Online]. Available: http://developer.android.com/reference/dalvik/system/DexFile.html

[23] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," 2003.

[24] B. Anckaert, M. Madou, and K. De Bosschere, "A model for self-modifying code," 2007.

[25] J. Bramley. Caches and self-modifying code. Visited: May, 2012. [Online]. Available: http://blogs.arm.com/software-enablement/141-caches-and-self-modifying-code/

A. Dynamic dex file loading

A.1. Java implementation for dynamic dex file loading

```
JNIEXPORT jint JNICALL Java_org_dexlabs_DexLoader_openDexFile
                       (JNIEnv* env, jobject thiz, jbyteArray dexdata)
{
        JValue pResult;
        jint result;
        u4 args[] = { (ArrayObject*)dexdata };
        dvm_dalvik_system_DexFile[1].fnPtr(args, &pResult);
        result = pResult.l;
        return result;
}

JNIEXPORT jobject JNICALL Java_org_dexlabs_DexLoader_defineClass
                            (JNIEnv* env, jobject thiz, jobject name,
                            jobject loader, jint cookie)
{
        StringObject* nameObj = (StringObject*) name;
        Object* loaderObj = (Object*) loader;
        u4 args[] = { nameObj, loaderObj, cookie };
        JValue pResult;
        dvm_dalvik_system_DexFile[3].fnPtr(args, &pResult);

        jobject *ret = pResult.l;
        return ret;
}
```

A.2. Native implementation for dynamic dex file loading

```
public class DexLoader extends Activity
{
     public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        InputStream is;
        byte[] buffer = new byte[2528];
        //fetching further dex file,
        //we can also do decryption at this point
        AssetManager assetManager = getAssets();
        is = assetManager.open("classes.dex");
        is.read(buffer);

        //load dex file into the process,
        //find a class by its name
        int cookie = openDexFile(buffer);
        Class newcls = loadClassBinaryName("com.protect.newclass",
                                           getClassLoader(),
                                           cookie);
```

```
19          ...
20      }
21      public Class loadClassBinaryName( String name,
22                                        ClassLoader loader,
23                                        int mCookie) {
24          return defineClass(name, loader, mCookie);
25      }
26
27      native private Class defineClass( String name,
28                                        ClassLoader loader,
29                                        int cookie);
30      native private int openDexFile(byte[] fileContents);
31      static {
32          System.loadLibrary("dexloader");
33          System.loadLibrary("dvm");
34      }
```

B. Self modifying Dalvik bytecode via JNI

B.1. Java implementation for self modifying Dalvik bytecode

```
1  public class ModiActivity extends Activity {
2      @Override
3      public void onCreate(Bundle savedInstanceState) {
4          super.onCreate(savedInstanceState);
5
6          Integer err = makemodification();
7          int egg = 0x23420023;
8          Integer result = 9;
9      }
10
11     native private  int makemodification();
12     static {
13         System.loadLibrary("hello-jni");
14     }
15 }
```

B.2. Native implementation for self modifying Davik bytecode

```
1  JNIEXPORT jint JNICALL Java_com_modi_ModiActivity_makemodification
2                                          (JNIEnv* env, jobject thiz)
3  {
4          //POC: correct values can be find by /proc/self/maps
5          char* start = 0x40037000;
6          char* end = 0x40039000;
7
8          //enable write access
9          mprotect( (void *)start, (end-start), PROT_READ|PROT_WRITE);
10
```

```
11          char * candidate = memchr(start, 0x14, (end-start));
12          //egg hunter
13          while( candidate != 0 && !(     candidate[2] == 0x23 &&
14                                          candidate[3] == 0x00 &&
15                                          candidate[4] == 0x42 &&
16                                          candidate[5] == 0x23 ) )
17                  candidate = memchr(candidate+1, 0x14, (end-candidate));
18
19          //find and manipulate  const/16 v4, #int 9 // #9
20          if( candidate != 0 &&    candidate[6] == 0x13 &&
21                                   candidate[7] == 0x04 &&
22                                   candidate[8] == 0x09 &&
23                                   candidate[9] == 0    )
24          {
25                  // const/16 v4, #int 265 // #265
26                  candidate[9] = 1;
27                  return candidate-start;
28          }
29          return 0;
30  }
```