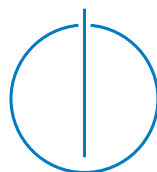
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Automotive Software Engineering

Research and Analysis of Copy Protection Mechanisms for Android Apps, as well as Implementing a Sample Application

Sebastian Schleemilch



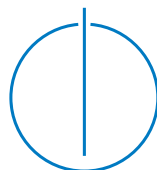
DEPARTMENT OF INFORMATICS
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Automotive Software Engineering

Research and Analysis of Copy Protection Mechanisms for Android Apps, as well as Implementing a Sample Application

Erforschung und Analyse von Kopierschutzverfahren für Android Apps, sowie Umsetzung in einer Beispielapplikation

Author: Sebastian Schleemilch
Supervisor: Prof. Dr. Uwe Baumgarten
Advisor: Nils Kannengießer, M.Sc.
Submission Date: 15.04.2016



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2016 Sebastian Schleemilch

Abstract

Google's Android has grown into one of the most popular mobile operating systems on the market also due to the rapidly increasing smartphone usage over the past few years. So called "Apps" can be installed through online stores; the most popular "Google Play Store" is being installed by default on most devices. Those Apps are being used for all kind of every day problems and even for online banking and other sensitive tasks. In order to protect intellectual property of used algorithms or to prevent insertion of malicious code, these apps need to be secured in terms of copy protection resistance as well as reverse engineering capabilities.

Static and dynamic obfuscation is a common technique to provide some barriers for attackers. Google's concept of distributing Apps through machine independent code (Dalvik Executables, DEX) and using Just-In-Time (JIT) compiling, simplifies the reverse engineering of Apps massively and opens an enormous gate for patching and repackaging.

This master's thesis presents new concepts in order to avoid App copying, patching and reverse engineering.

Android's runtime architecture changes very frequently and recently from a Dalvik Virtual Machine JIT concept to a new Android Runtime (ART) using Ahead-Of-Time (AOT) compiling. Those changes are affecting the use of DEX and its optimized native code version (OAT). Therefore, they are investigated with regard to their copy protection domain.

A great part deals with dynamic code loading using JNI possibilities and its corresponding copy protection applications. It complicates the reverse code engineering of Apps enormously, but cannot avoid it completely. Another concept are Trusted Execution Environments that do create a trusted world. Unfortunately due to a high fee, it is not accessible for most developers and therefore not extraordinary useful for common copy protection techniques.

The possibility of distributing native code, like being routine in desktop environments, will also be analyzed.

This master's thesis provides application examples regarding dynamic code loading, natively as well as in the Java world. They address concrete solutions like string encryption and licensing

improving. Since there is an AOT compiling step, the performance is nearly independent from its implemented language.

Developed concepts in this master's thesis provide a general understanding of different copy protection approaches in conjunction with ART.

Contents

Acronyms VIII

1 Introduction 1

1.1 Popularity of Android . . . 1

1.2 Demand of App Protection Mechanisms . . . 1

2 Fundamentals/Basics 4

2.1 Android Architecture . . . 4

2.2 Android Apps . . . 6

2.3 App Installation Process . . . 8

2.4 App Execution Process . . . 10

2.4.1 Dalvik . . . 10

2.4.2 ART . . . 10

3 Copy Protection Status Quo 12

3.1 ART Internals . . . 12

3.1.1 APP Executable Format . . . 12

3.1.2 App Execution . . . 19

3.2 DEX Disassembly and Repackaging . . . 25

3.3 Obfuscation Techniques . . . 26

3.3.1 Static . . . 27

3.3.2 Dynamic . . . 28

4 Android Dynamic Native Code 33

4.1 NDK and Android Studio Integration . . . 33

4.2 Dynamic Shared Library Loading out of a File . . . 35

4.3 Dynamic Binary Execution out of a File . . . 38

4.3.1 Java Implementation . . . 39

4.3.2 C/C++ Implementation . . . 40

4.4 Dynamic Code Execution from Memory 41
4.4.1 Android Memory Mapping 41
4.4.2 Code Execution 43
4.5 Performance Comparison 46
4.6 General Memory Access 47
4.7 Utilizations 48
4.7.1 Licensing Improving 49
4.7.2 String Encryption 51
4.8 Decompilation 55

5 ART Native Code Store 58

6 Trusted Execution Environments (TEEs) 61

7 Related Work 64

8 Conclusion 67

8.1 Future Work 68

Appendices i

A NDK Project ii

B NDK AES Implementation v

Acronyms

ABI Application Binary Interface.

ADB Android Debug Bridge.

AOSP Android Open Source Project.

AOT Ahead-Of-Time.

API Application Interface.

APK Android Application Package.

App Short term for (mobile) application.

ART Android Runtime.

ASLR Address Space Layout Randomization.

CPU Central Processing Unit.

DEX Dalvik Executable.

DVM Dalvik Virtual Machine.

ELF Executable Linking Format.

GID Group Identification Number.

GLIBC GNU C Library.

GUI Graphical User Interface.

HAL Hardware Abstraction Layer.

IDE Integrated Development Environment.

IPC Inter Process Communication.

JAR Java Library.

JDK Java Development Kit.

JIT Just-In-Time.

JNI Java Native Interface.

JVM Java Virtual Machine.

MIME Internet Media Type.

NDK Native Development Kit.

ODEX Optimized Dalvik Executable.

OOP Object Oriented Programming.

OS Operating System.

PID Process ID.

PIE Position Independent Executable.

pm Package Manager.

PPID Parent Process ID.

REE Rick Execution Environment.

SE Secure Element.

SoC System-on-Chip.

TCG Trusted Computing Group.

TEE Trusted Execution Environment.

TIS Tool Interface Standards.

TPM Trusted Platform Module.

UI User Interface.

UID User Identification Number.

VM Virtual Machine.

ZIP Zipper file format.

Chapter 1

Introduction

1.1 Popularity of Android

Android is an operating system by Google, designed for mobile devices. Version 1.0 was released in 2008 and the most recent version is 6.0 as this thesis is written. With a remarkable market share of 82.8% it has grown into the most important mobile operating system followed by Apple's iOS with 13.9% and Microsoft's Windows Phone with 2.6%. See [Inc15] for a full list of mobile operating system market shares. While the smartphone market is still growing, Android did also adapt to new rising platforms like wearables, TV's, cars and to embedded devices in general. Android is built on a Linux kernel and because of that, Android is open source and freely available. Everyone is allowed to adopt it to it's own needs (except Android Wear) which is another reason for it's rising popularity.

The daily use of smartphones and therefore Android is increasing rapidly in nearly every field of application. Right now, the usages are reaching from simply surfing the web to security sensitive tasks like banking transactions, the organization of all kinds of tickets and wireless payment methods. One of the main factors of success are App Stores who provide simple installable applications for nearly every imaginable task so far.

1.2 Demand of App Protection Mechanisms

The popularity of operating systems generally result in an increasing interaction between developers and a large variety of Apps. At the same time it attracts malware and virus developers, which try to exploit the system due to its market size and the resulting high possible illegal profit. Although Google has established their own App Store "Google Play Store" where Apps are

getting inspected by Google's "Bouncer", it is possible to install Apps from so called "unknown sources" like websites and alternative App Stores.

There are different attack vectors that can be considered by attackers. The main ones are summarized into three following scenarios:

Scenario 1: Piracy/Intellectual Property Theft

Something that is present in every domain where program code is being written and can be sold, is illegal copying of a whole executable program or copying and adapting its source code. Not every Android App is available for free. So being able to copy an App to another device and get it to run is a serious threat for developers. However, there do exist licensing mechanisms to check if a user is authorized to use the App. If he is not, the App can be exited. Another possibility for attackers would be to disassemble the App into its source code to copy and adapt the App. Afterwards, the attacker could release it on its own (for instance with a different name and layout). This scenario is hazardous for Apps that do contain a lot of know-how in form of written code.

Scenario 2: App Patching

Many Apps these days are free of charge in their basic version and do offer "In-App Purchases" for additional content or services. An attacker could try to circumvent the licensing mechanism by "patching" the App. This means injecting code at runtime or trying to obtain its source code followed by adding code with the goal to bypass the implemented licensing mechanism. The attacker can then repackage and reinstall the changed App with extended access rights. This leads to a financial loss for developers.

Scenario 3: Malicious Code Injection

Quite similar to scenario one, the attacker chooses a popular App he wants to exploit but this time the focus lies on getting sensitive data from users. So a good example would be a banking App where users can do transactions and have to enter pins and TANS. To be able to sniff sensitive data, one possibility is to get the App's source code like in scenario one and injecting own code that implements the malicious functionality. After repackaging it, potential users need to be tricked into installing the App, which appears to be the official banking App. This can be done using common social engineering techniques since it should not be possible to upload the App to an official App store if the original App is present.

Therefore there is an urgent need of copy protection mechanisms for Android Apps. This basically means in most cases, to prevent reverse code engineering of the distributed Application packages. The goal is to hinder the repackaging of an App that includes malicious or generally altered code.

Chapter 2

Fundamentals/Basics

2.1 Android Architecture

To provide copy protection mechanisms for Android Apps, a deep understanding of the Android system itself is necessary or at least helpful. Android is built on the Linux kernel. Android does provide not only shell binaries (Linux kernel) but also a GUI environment as well as predefined frameworks. It offers a complete environment for developers to write Apps in the Java programming language. That's why Android is considered to be a "full software stack" [Lev15, p.7f]. Although it's built on Linux, Google did modify the Linux kernel according to their needs. As a result, the Android kernel differs and is incompatible with the Linux kernel since version 2.6.27.

The difference at kernel-level is not that big compared to the differences at user-mode where Android does have entirely new components, which are:

- Dalvik Runtime including the Dalvik Virtual Machine (DVM) respectively the Android Runtime (ART) since Android version > 5.0.
- The Bionic C-Library instead of the GNU C Library (GlibC)
- Hardware Abstraction Layer (HAL)
- Java Native Interface (JNI)
- Android specific frameworks

Figure 2.1 visually summarizes the difference between Android and Linux. The Android specific frameworks are the core components that make Android special. They simplify the creation process of applications massively. Developers can use the higher level language Java rather than

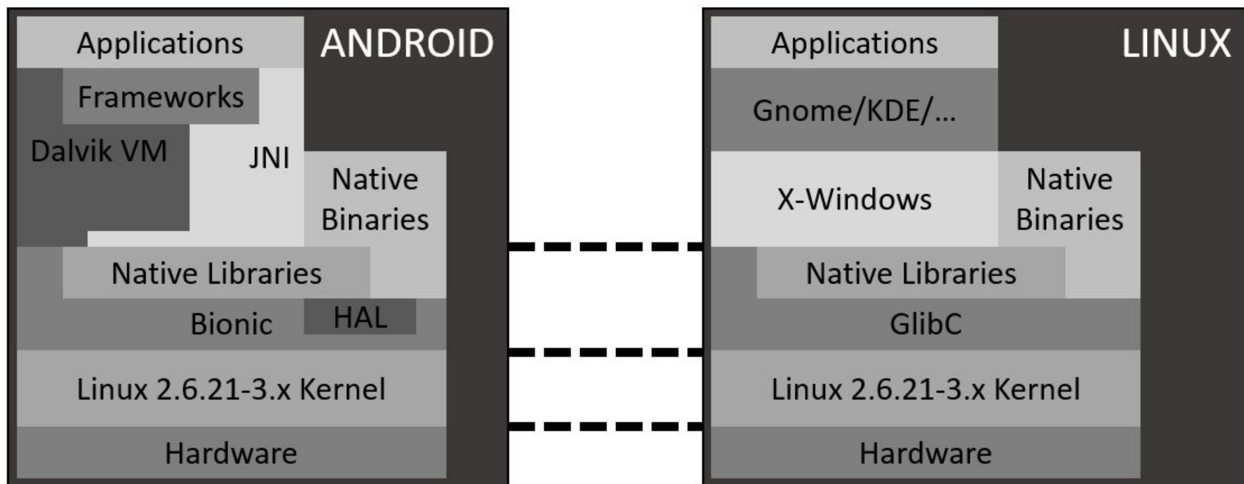


Figure 2.1: Android and Linux comparison taken from [Lev15, p.9]

developing in C/C++. Additionally, there is a rich set of APIs to solve most of programmers everyday problems.

In order to get Java programs run on Android, the DVM is introduced which is quite similar to a Java Virtual Machine (JVM) but more simple. The reason for choosing a virtual machine in the first place was due to the limited storage capacity of mobile devices, especially in their early days [Lev15, p.11f]. The DVM provides an interface between the operating system and the Java application world, after all the execution of Java written programs. The difference between the DVM and the JVM is the alternative form of bytecode which gets executed in the VM. Another difference is that Dalvik is register-based in contrast to JVM's stack based architecture. DVM is optimized for mobile devices in terms of efficiency and sharing memory. It uses device independent Dalvik Executable format files (DEX), which is one reason for choosing a VM design in the first place [Lev15, p.11f]. DEX files can be packed into Java libraries (JAR) or into Android Application Packages (APK). They will get created automatically by the official Android Studio IDE from Google with Java code as its source.

Generally, a runtime's purpose is to provide interpretation of machine independent code, in other words, transforming Java byte-code into machine-code (pure 1's and 0's a CPU understands). With Android version 5.0, Google did introduce ART which is an alternative runtime. Like Dalvik, it does use a VM but with a different compile timing concept. The prior Dalvik runtime does use Just-In-Time (JIT) compiling where executable machine code is not created before the App runtime. ART on the other hand, uses Ahead-Of-Time (AOT) compilation which compiles an App to machine code at installation time. As a result, the ART Apps installation time as well as

the required storage space are increased. This is a trade-off between the App startup time and runtime performance.

The JNI provides an opportunity to write Java programs with embedded native processor specific code to escape from the VM world to, for instance, access hardware directly. It is mostly used to optimize the performance for Apps (e.g. games) or to impede reverse engineering of the application code. Google does provide a Native Development Kit (NDK) to help developers create native libraries [Goo16b]. It includes cross compiling toolchains for all supported Android architectures. The officially supported language for the NDK is C/C++.

Bionic is the Android corresponding GlibC library which was created for license and simplicity reasons. Overall, it is more lightweight than the GlibC and well adapted for Android's needs.

Since Android is likely to run on a great variety of devices, it has to support a big amount of different hardware versions. The HAL addresses this problem and standardizes the interface by allowing hardware vendors to implement their own drivers [Lev15, p.18f]. It helps developers to use the same programming interface to hardware even when the device architecture and hardware components differ.

2.2 Android Apps

Apps are the most abstract level of the Android software stack. Generally, there are either system-apps (mostly pre-installed, like a browser) or user-apps (everything installed by user). The system-apps storage location (`/system/apps/`) differs from the user-apps (`/data/apps/`). System-apps are included into the distributed OS whereas user-apps can be installed and updated via the Android Debug Bridge (ADB), App stores like the Google Play Store or directly on the device by opening downloaded App files (APKs). The most common way for consumers is downloading/installing Apps via an App Store.

Every App has its own sandboxed environment. That means that Apps cannot access data from other Apps and they can only access resources those permissions were granted during installation time. This is one of the basic Android security concepts, the privilege separation and the principle of the least privileges [Ele15, ch.1] which can be applied through Linux group and user permission concepts.

Apps do have several components, the main ones are:

- Activities
- Services

- Content Providers
- Broadcast Receivers

Activities do represent a screen view in the UI ready for user interaction. They can be started independently even though one Activity is chosen to be the one that starts when the user clicks on the App icon. Services are purely functional and are not represented at the UI compared to Activities. Their utilization are mostly time consuming operations in the background of an App like server communication or computational tasks.

Since Apps are sandboxed, they can't access each others data which can be seen as a limitation if Apps should be able to communicate with each other. That's why Content Providers offer an interface to the App data which is chosen to be exchanged. They use a lower level Inter Process Communication (IPC) protocol provided by the Android Stack which, like the name indicates, operates on process level in Linux. With Broadcast Receivers, an App can react to system-wide events (broadcasts) [Ele15, ch.1].

Apps are distributed in form of machine independent `.apk` files. The APK file is a file container and is an extension to JAR which is in turn an extension to the ZIP file format. Thus, the content of the APK can be extracted with standard ZIP decompression tools [Ele15].

Figure 2.2 lists the content of an APK container after extraction.

The `AndroidManifest.xml` file holds meta-data information about the App like the package name, its version, a list of activities (one of them is marked as the action `MAIN` activity) and required permissions. `resources.arsc` contains precompiled resources like strings, styles or binary XML. Developers can use the `assets/` folder to store raw data needed for the App like music files, fonts, additional DEX files and every other arbitrary data. If the App does make use of the JNI, the `lib/` folder contains compiled library directories for every supported CPU architecture (armeabi, x86, ...). Every resource which is directly addressed from Android code, is stored in the `res/` folder including XML files for layouts and menus. The `META-INF/` directory holds the signatures for the specific App, similar to a JAR file. The heart of every App is the `classes.dex` that contains the application code in form of the machine independent DEX format specified by Google [Goo16d] which is comparable to Java byte code. The Java Development Kit (JDK) tool "`javac`" creates a JAR out of every inputted `.java` file while the Google build tool "`dx`" takes over the transition from `.jar` to `.dex` [Goo16a]. An `ApkBuilder` takes the DEX file, adding `.so` files and other resources and finally builds the `base.apk` that gets signed afterwards by the `jarsigner`.

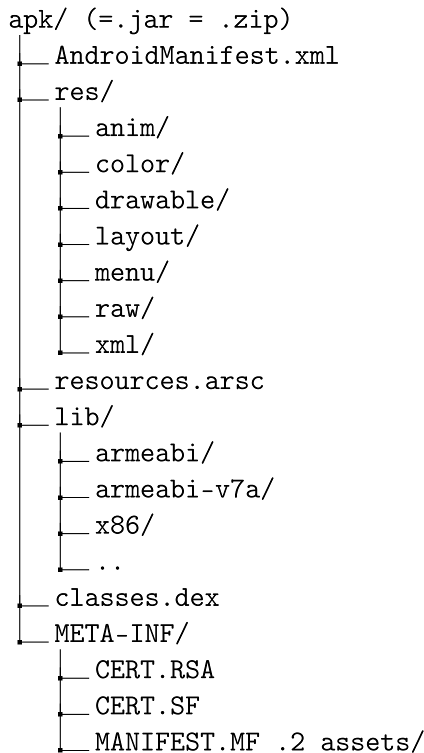


Figure 2.2: APK Content Tree

2.3 App Installation Process

As in section 2.2 described, Apps are distributed in form of APK files. This chapter describes what happens next with the APK file.

The APK file gets forwarded to the Android "Package Manager"(pm) which then calls the installPackage() method for this APK. Among other things, the pm copies the original *.apk to the path /data/app/<package>.<appname>-1/base.apk. This location is used in order to get the needed resources for the App to run (layout, drawables, ...). The lib/ directory is extracted and gets copied into that path. classes.dex is optimized by either "dexopt" or "dex2oat" depending which runtime is present (DVM or ART). In both cases, the outcoming file is no more machine independent and is stored at /data/dalvik-cache/<arch>/ with the name data@app@<packagename>.<appname>-1@base.apk@classes.dex. That path however, did change with Android version 6 to /data/app/oat/<pachagename>.<appname>/<arch>/base.odex. It is more straight forward since the whole data of an App is being stored at the same place (except for the internal storage an App can use, which is located at /data/data/...). It does represent the last stored unit of an App that will be executed in the next step. Attention should be paid to the file extension. It is the same (.dex) with both runtimes but is actually a totally different file format. Therefore, the name of the file is just a reference to the path of its source when

replacing the "@" characters with "/" (prior Android 6). Since Android 6, it became a little more convenient since the file extension changed to "`.odex`" which corresponds to its actual content since it is an optimized version of the DEX. If the present runtime is DVM, thus Android < 5.0, the generated file format is ODEX (Optimized DEX) and in later versions ELF 32/64 (Executable Linking Format) although the responsible tool for that conversion step is called "`dex2oat`". Still, the file is named the same way and won't change to "`.elf`" so the naming convention is more like a semantic information rather than making an explicit statement about its content. This ELF file will be analyzed later in this thesis (it does however hold an "OAT" file among other things so that the tool name actually does makes sense). To give an App the possibility to store some data, a `/data/data/<appname>` directory is finally created which is a storage whose permissions are limited to its owning App.

Last but not least, the pm adds entries to the `/data/system/packages.xml` as well as the `/data/system/packages.list` files. They do contain meta information about the installed packages like required permissions and the UID/GID of that instance. Android uses that UID/GID to enforce the sandboxing model [Ele15, ch.1]. Figure 2.3 shows the explained relevant actions the pm performs.

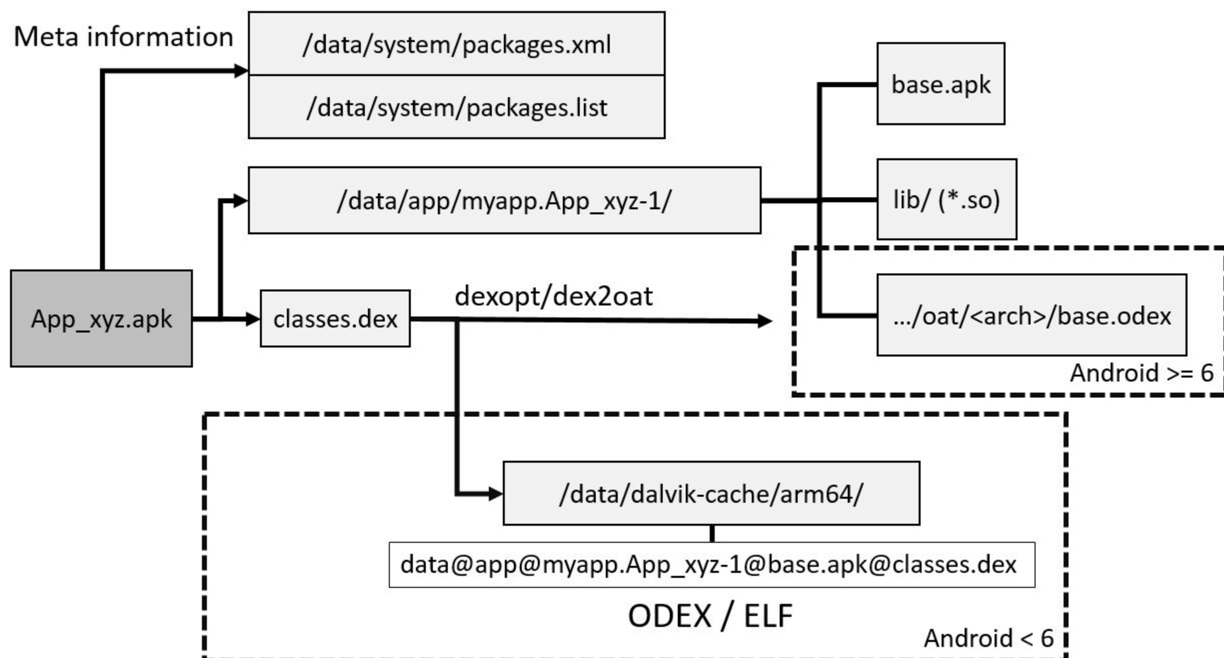


Figure 2.3: App installation process

2.4 App Execution Process

This chapter is supposed to give a short introduction to the main concept of the App execution process, without going into great detail about it. However, a detailed inspection will be performed in subsection 3.1.2. When describing the execution process of an App, again the runtime must be considered (Dalvik or ART) since the execution process obviously differs. In both cases, the entry-point of the App is the file stored at /data/dalvik-cache/...@classes.dex as described in section 2.3 (Again differs depending on the Android version). "Zygote" is the name of one of the first processes Android starts when booting. The process is responsible for loading more services and libraries of the Android framework and holds precompiled resources that nearly every App instance will need. By starting an App, it is therefore much more efficient to fork the Zygote process on the Linux layer and to include the App specific parts afterwards rather than creating a whole new process for every App. This will also be investigated in greater detail in section 3.1 [JDr14, ch.2].

2.4.1 Dalvik

As in section 2.3 described, an ODEX file is created at the first startup of an App. Even though the pure DEX file is runnable in the DVM, it still gets optimized into ODEX to get the most performance out of the VM constraints. It then gets processed by the actual Dalvik runtime to produce native code (machine code) that in the end is executable by the CPU (JIT). This native code has to be linked with libraries of the App (/data/app/<pkg>.<name>-1/lib/*.so) implemented via the JNI and additionally with Android framework libraries to result in a complete executable that can be executed by the CPU. It does reference resources out of the corresponding base.apk file (layouts, drawables, assets, ...). The execution process is also depicted in Figure 2.4.

2.4.2 ART

The difference of ART in comparison to Dalvik is the missing JIT compiling step which ART performs at installation time (AOT). Therefore, the ELF file contains the executable that only needs to be extracted by the ART followed by the same linking steps that Dalvik does after JIT. Figure 2.4 shows a direct comparison of those two runtimes.

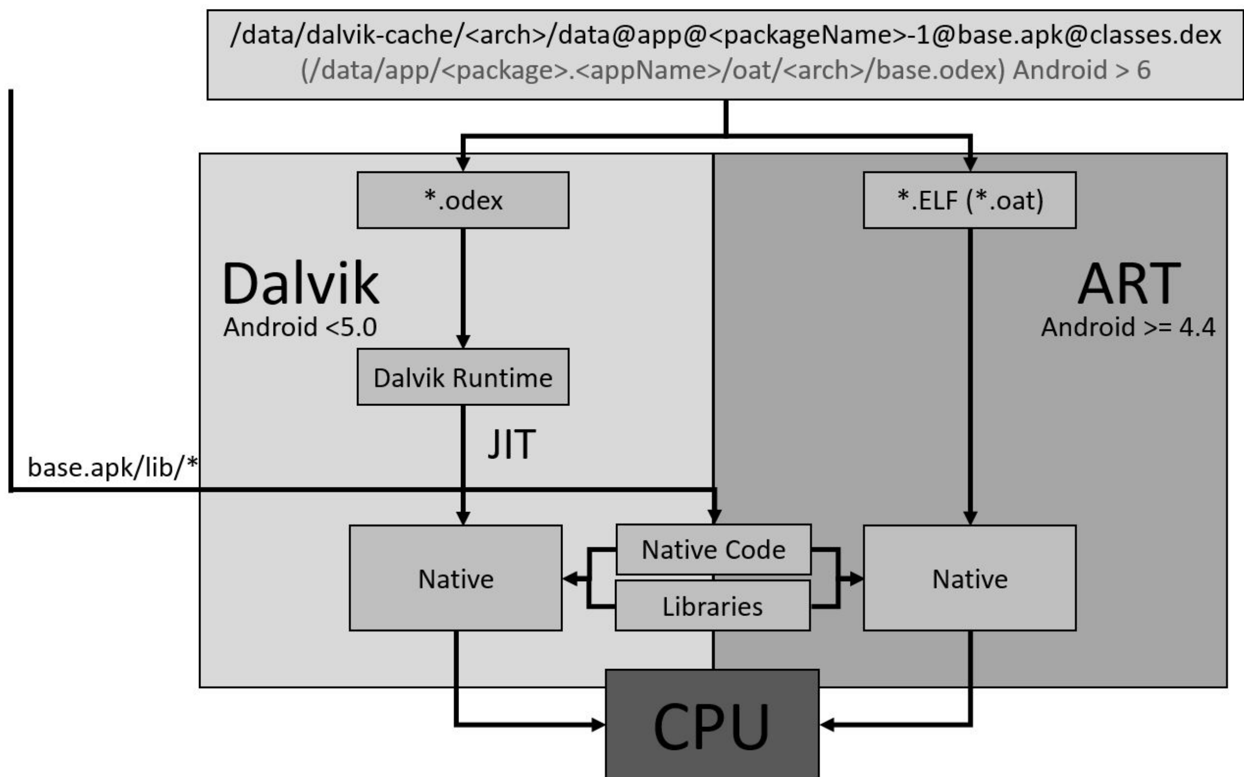


Figure 2.4: App execution process

Chapter 3

Copy Protection Status Quo

This chapter will first provide some detailed information about the ART runtime since basic knowledge of the internal mechanics is a precondition to understand the more complex copy protection mechanisms and obfuscation techniques. Common existing techniques will be sketched afterwards. Most of them, are based on the Dalvik runtime. It will then be determined whether a specific method is also applicable to ART. A great focus will rely on the DEX format since it is still the distribution format of every App (inside of an APK named as `classes.dex`). Unless explicitly written otherwise, investigations performed do rely on an AOSP and device version running Android 5.1.1. Therefore, they may vary on different Android versions since Google applies changes at runtime layer very frequently. Those changes usually do not affect App developers who are writing code at Java layer but may affect developers using the NDK.

3.1 ART Internals

3.1.1 APP Executable Format

A core element of the recently introduced ART is the file that gets created by "`dex2oat`" during the installation time of an App, described in section 2.3. Since ART does use AOT compilation, the file format is expected to be an executable or at least a native code container. Therefore, it is worth to have a closer look at that file format especially since Google does not provide any further information about its content. It might have the potential of revolutionizing the available copy protection mechanisms for Android or having at least a great impact.

By applying the Unix command "`file`" (which can classify files to MIME-types) to the resulting file of the "`dex2oat`" tool it comes apparent that it is a particular ELF file (32 or 64 bit) called OAT file from now on.

ELF File Format

ELF was originally specified by UNIX System Laboratories (USL) and later by Tool Interface Standards (TIS) and is a common standard for executables, object code and shared libraries on UNIX systems. It is a quite flexible format for different CPUs and architectures and serves as a container for different executable binary formats.

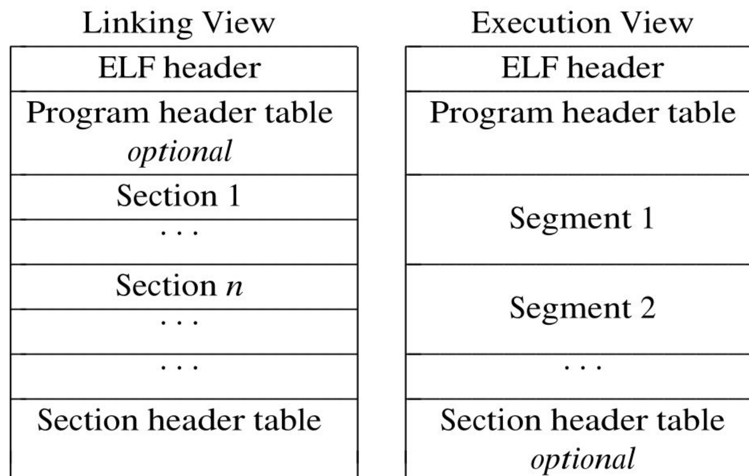


Figure 3.1: ELF file format taken from [Com93]

Figure 3.1 shows the file structure. One has to differentiate how this file is viewed based on its context. While a linker does care about sections, sections may be glued together to segments when executing the file. Meta data about the file can be read out of the "ELF header" that starts at address `0x00` and contains information about the version, file type, target machine and offsets to the program- and section header tables. In ART, the file is marked as an shared object with LSB encoding and not as an executable. That makes clear, that this file is not supposed to get executed directly, but to be linked first (An open question that remains so far is which process is responsible for the App start). Segments are referenced by the program header table and sections by the section header table. For an execution process, only the header and the information out of the program header table is needed [Kov12].

Let's first have a look at the used sections in case of the specific Android implementation, the OAT file: Table 3.1 shows the output of "`readelf -S <ELF-App-File>`", listing all available sections.

It does follow a short description of sections that are implemented [Kov12]:

- `.dynsym` holds a dynamic linking symbol table that contains information for locating and relocating a program's symbol definitions and references. It contains "oatdata", "oatexec" and "oatlastword" in case of an OAT file.

- `.dynstr` holds strings for dynamic linking, mostly names that are referenced by `.dynsym`.
- `.hash` is a hash table that supports symbol table access
- `.rodata` stands for "Read-Only data" and contains arbitrary data whose interpretation is solely determined by the program itself. We will see that in case of Android it does hold the actual OAT file that will be further described in section 3.1.1.
- `.text` is the only region that is marked as executable and holds the main body of program code.
- `.dynamic` includes dynamic linking information.
- `.shstrtab` stands for "Section header string table" and contains the previous described section names including its own (e.g. "`.shstrtab`").

Table 3.1: ELF section headers

Secion Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.dynsym	DYNSYM	000000d4	0000d4	000040	10	A	2	0	4
[2]	.dynstr	STRTAB	00000114	000114	000029	01	A	0	0	1
[3]	.hash	HASH	00000140	000140	000020	04	A	1	0	4
[4]	.rodata	PROGBITS	00001000	001000	002000	00	A	0	0	4096
[5]	.text	PROGBITS	00003000	003000	000228	00	AX	0	0	4096
[6]	.dynamic	DYNAMIC	00004000	004000	000038	08	A	1	0	4096
[7]	.shstrtab	STRTAB	00000000	004038	000038	01		0	0	1

Table 3.2 shows the alternative view on the file by having a look at segments ("`readelf -l <ELF-App-File>`") . Type "PHDR" stands for "Program header". Segments with type "LOAD" are supposed to be loaded from disk into memory while a "DYNAMIC" segment is a part of a "LOAD" segment and is equal to the "`.dynamic`" section. The mapping of "LOAD" segments into memory is performed by respecting the alignment of `0x1000` which means that only chunks of that size (or a multiple) are being read (e.g. reading the segment at `0x3000` will copy the content from `0x3000-0x4000` even if the size only equals `0x340`). The difference between "FileSiz" which stands for the file size and "MemSiz" which stands for memory size, is the space that is reserved for uninitialized variables.

Table 3.2: ELF program headers

Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00000034	0x00000034	0x000a0	0x000a0	R	0x4
LOAD	0x000000	0x00000000	0x00000000	0x03000	0x03000	R	0x1000
LOAD	0x003000	0x00003000	0x00003000	0x00340	0x00340	R E	0x1000
LOAD	0x004000	0x00004000	0x00004000	0x00038	0x00038	RW	0x1000
DYNAMIC	0x004000	0x00004000	0x00004000	0x00038	0x00038	RW	0x1000

The "`readelf`" tool is also capable of showing the resulting mapping of sections to segments (Table 3.3).

Table 3.3: ELF section segment mapping

00				
01	.dynsym	.dynstr	.hash	.rodata
02	.text			
03	.dynamic			
04	.dynamic			

It is interesting to note, that the Android usage of ELF for Apps is very minimalistic and contains very few sections/segments compared to a common program written in C/C++ (`helloWorld.c` does include over 30 sections). As described before, `.dynsym` contains entries which tell us where to find the OAT data, specifically the "oatdata"(equals `.rodata`) and the "oatexec" (equals `.text`) section that will be analyzed now.

OAT File

Google does not provide any official documentation about the OAT file format other than the source code itself (`art/runtime/oat[_file].h[c]`). [Sab15] however gives a helpful introduction. An overview is given in Figure 3.2 that shows the content of "`oatdata`" and "`oatexec`" which are scattered among the superordinate ELF sections.

Important attributes that the "OAT Header" contains are the checksum over itself, the instruction set (ARM, ARM64, MIPS, . . .), the native code offset relative to the beginning of "`oatdata`" (located in "`oatexec`") and the quantity of embedded DEX files (in case of Apps it always equals one, other

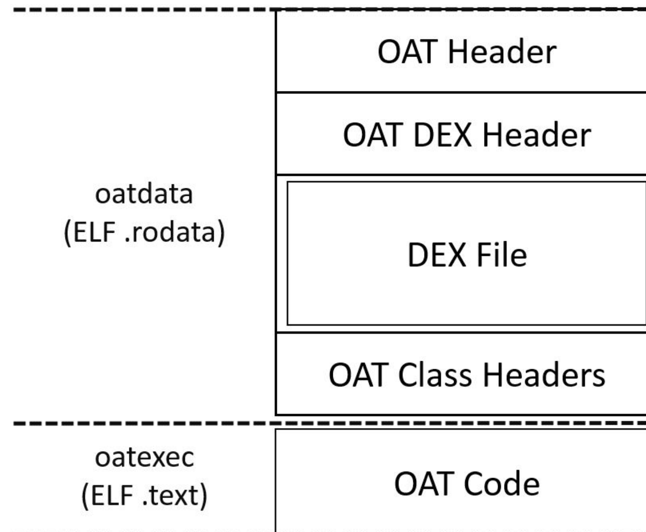


Figure 3.2: OAT format

system files like "boot.oat" may contain several DEX files). It follows the "OAT Dex Header", containing the absolute path of its "source" file (=input file of "`dex2oat`", means DEX), the checksum, the embedded DEX file's offset as well as an offset to the "OAT Class Headers". Those "OAT Class Headers" do offer information about defined classes. It includes the type of class, indicating how many methods in the class are compiled to native code (can be "all", "some" or "none" but should be "all" in almost every case) and additionally the offsets from the beginning of native code of every compiled method. The actual code is located in a superordinate section `.text`, as already explained, labeled as "`oatexec`" which is separated but referenced from "`oatdata`".

DEX File Format

For the sake of completeness, a description and explanation of the DEX format is also given, which is officially documented by Google [Goo16d]. Before the introduction of ART, DEX was the last unit before execution of an App (besides ODEX which can be easily converted back to DEX). The DVM however, accepts both formats and therefore, it is possible to execute DEX files without the optimization step. As a consequence, DEX files, which are not a part of the distributed App, can be loaded dynamically at runtime. That enables new possibilities for dynamic code obfuscation techniques that will be described in section 3.3. DEX as it is, does not only contain VM instructions, but also some meta data to locate higher abstraction level sections of the file like classes, methods and fields. Figure 3.3 shows the file layout. The header includes a checksum of the whole file (checksum field excluded), the overall size as well as the offsets and sizes of every section. Sections with "ids" ending are arrays of id type items and reference a data item

header
string_ids
type_ids
proto_ids
field_ids
method_ids
class_defs
data
link_data

Figure 3.3: DEX format

in the "data" section or are specifying an index to another "ids" section. A "string_id" item for instance, just contains an offset from the file beginning that should be located in "data" whereas the "type_id" item includes an index into the "string_ids" field. So every attribute that can be described as a string is concentrated in "string_ids". The principle of the file structure is therefore a architecture of referencing sections.

A part that puts everything together is the "class_defs" section. Classes in this context stand for object oriented programming classes. They include methods, which in turn include strings, fields and so on. Actual content like strings, VM instructions or arbitrary programming data, is being stored exclusively in the data section. In the end, executable instructions are referenced from encoded methods and defined in "class_data_items" which in turn are referenced from "class_defs".

Conclusion

The runtime transition from DVM to ART had to result in a new file that is interpreted/executed when an App starts. However, the change is not as smooth as expected since the new file format is not pure executable code but, as explained, a combination of compiled native code and embedded DEX code as well as a new OAT file format which references parts of the native code. Also, there is some confusion for what part the name "OAT file" stands for. On the one hand, Google's documentation files and the naming convention of the "`dex2oat`" tool are giving the impression that the file as a whole is an "OAT file". On the other hand, the file is a valid implementation of the well known ELF and contains a section that starts with bytes known by MIME types with ".oat" as file format (.rodata section). Additionally, the "oatexec" section is controlled via the hierarchically higher ELF. Therefore, "OAT file" most likely stands for for both, the Android App specific and minimalistic implementation of the ELF as well as the combination of the "oatdata"

and “oatexec” section to a file alike entity. Figure 3.4 provides an overall view of the file format of ART App executables.

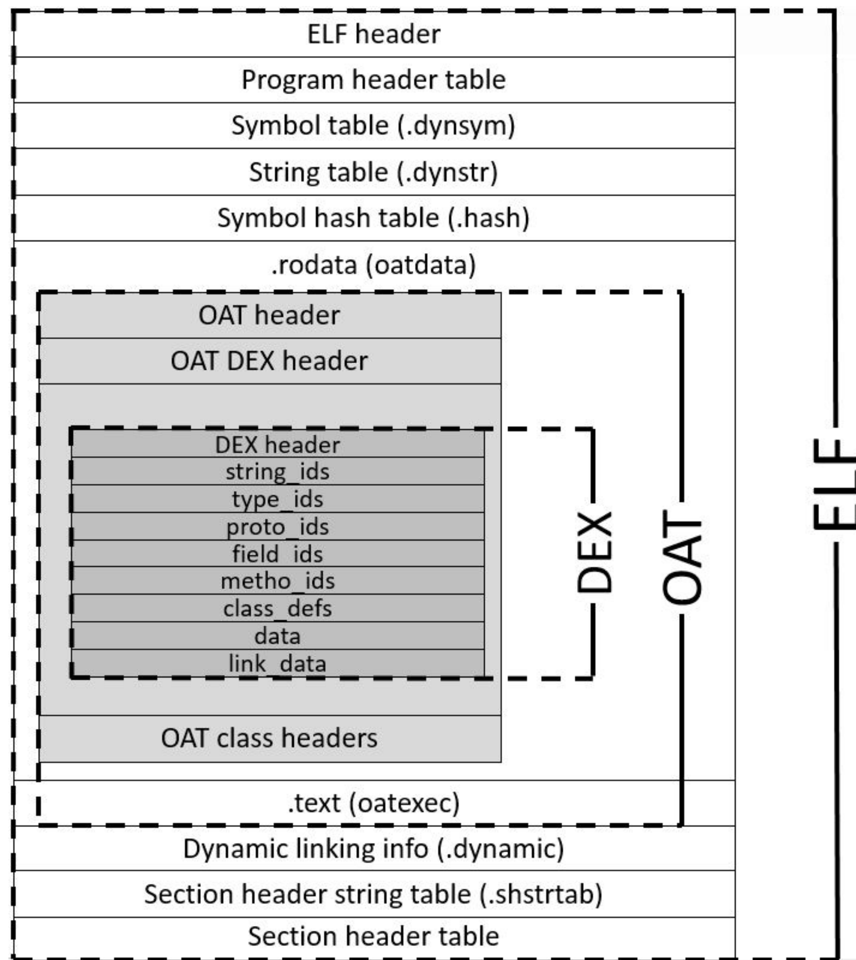


Figure 3.4: ART App executable format

The awareness of the detailed executable file format pops new questions about the ART internal functioning. Since the file is marked as an shared object, it will not be executed as a standalone program but most likely, invoked in a forked Zygote process like in the Dalvik runtime. However, it is not clear which parts of the OAT file are needed to run an App adequately. Is the embedded DEX for instance mandatory for the App to work correctly? As it will be described in a later chapter, DEX files are a crucial part of applications to protect. What if that part can be left out by distributing a minimal stub application followed by a dynamic native code injection at runtime? It would also be interesting to elaborate if and how common obfuscation techniques designed for Dalvik, described in section 3.3, can be applied to ART. A deep understanding for the App initialization and execution process under ART, is a precondition to answer those questions.

Therefore, a more detailed explanation of that process than previously described in section 2.4 follows.

3.1.2 App Execution

To get behind the scenes of the App execution, one can start at the resulting Linux process that exists for every running App. The Linux tool "`ps`" can be used to display running processes and therefore running Apps. To investigate the App execution, a root shell at the target device has to be opened (device should be rooted) with "`adb shell`" followed by an "`su`" command after getting the device prompt ("`shell@flounder:/ $`"), where `adb` is the Android Debug Bridge to interact with connected devices. The prompt will change to "`root@flounder:/ #`", signaling that it's a root shell. A "`ps`" command displays process information like its "USER", the numerical process id "PID", the process id of its parent "PPID" and of course the process name. Interesting entries for further inspection are being displayed in Table 3.4, showing the `init`, Zygote, the chrome and a sample "hello world" application.

Table 3.4: Android processes

USER	PID	PPID	...	NAME
root	1	0	...	/init
root	211	1	...	zygote64
root	212	1	...	zygote
u0_a137	10072	211	...	ma.schleemilch.helloandroid
u0_a35	11017	212	...	com.android.chrome

The process "`/init`" is the first process of Android (although it has a parent with PPID "0" which is the process scheduler at kernel level). It can be derived that every user and system App has either the process "`zygote`" or "`zygote64`" as its parent process if the App can be converted at the "`dexopt`" step to 32 or 64 bit. That makes clear, that Apps are forked from the Zygote process which in turn is a fork of "`/init`". Even more detailed information about processes can be pulled out of the "`/proc`" directory (an abbreviation for "process"). The directory offers an interface to the kernel space and does contain a folder for every process, named after its PID [Kal16]. The most attractive attribute for this investigation purpose of that folder is the "`exe`" attribute. It is a symbolic link to the executable that started the process. Since Apps are a fork of Zygote, they should also point to the same executable, which they do (see Table 3.5).

Table 3.5: Process starting executables

`root@flounder:/ # ls -la /proc/10072/exe`   `...  exe -> /system/bin/app_process64_original`
`root@flounder:/ # ls -la /proc/211/exe`   `...  exe -> /system/bin/app_process64_original`
`root@flounder:/ # ls -la /proc/11017/exe`   `...  exe -> /system/bin/app_process32`
`root@flounder:/ # ls -la /proc/212/exe`   `...  exe -> /system/bin/app_process32`

An executable named "`app_process32/64`" seems to be the entry-point for Apps (or at least Zygote) to get started. The responsible program of that executable can be found in the Android Open Source Project (AOSP) where "`app_main.cpp`" is the name of the source code file that gets compiled into the 32 or 64 bit version and can be found at "`/frameworks/base/cmds/app_process/`". Although its implementation allows Apps to get started directly without Zygote, they are not supposed to, meaning it's not the common way when a user clicks on an App icon. Instead, it will get clear that Zygote is capable of forking and transforming itself into a new App. The App process is therefore responsible for starting Zygote and other system related processes that have a similar structure like Apps. The "`/init`" process arranges the start of Zygote via an "`app_process`" program start with specific parameters for a Zygote start (init is responsible for far more, like starting native daemon system services and other things).

A common way to analyze source code is to start with its "`main()`" method. One of the first things the program does is creating a new "`AppRuntime runtime`" object that inherits from the `AndroidRuntime` class but overwrites a few functions. As parameters, it will expect the `argv[0]` which is the program name itself, as well as the total length of arguments. In case of a Zygote start, the program will transfer the flow control to this object by calling `runtime.start(com.android.internal.os.ZygoteInit, args)`. "`args`" contains information about whether to start the system server,an ABI list as well as remaining arguments that were not used for the `app_process` program. The runtime `start()` method initializes a VM and finally calls the main method of the `ZygoteInit.java` program in case of a Zygote start. So there is a first transition into the Java programmed system.

Zygote

`ZygoteInit's` main starts with parsing arguments such as if a system server should get started, the ABI list and the socket name that Zygote will create. That defined socket (`LocalServerSocket sServerSocket`) is a core functionality of Zygote whose purpose is to listen to a socket for App start requests and finally fork Zygote and specialize it. After registering that socket, a preloaded method loads classes resources, OpenGL and shared libraries and a explicit garbage collection `gc()` will be performed to clean up after startup. After the optional start of the system server, the program will enter its main loop `runSelectLoop(abiList)`. Garbage collection is called explicitly after `GC_LOOP_COUNT` iterations. Right before entering a loop, an array list for file descriptors (`ArrayList<FileDescriptor> fds`) as well as for socket connections (`ArrayList<ZyogteConnection> peers`) are being created and `fds` gets filled with the server socket file descriptor. Listing 3.1 displays the logic of the main loop.

Listing 3.1: ZygoteInit main loop

```
            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray);
            } catch (IOException ex) {
                throw new RuntimeException("Error in select()", ex);
            }

            if (index < 0) {
                throw new RuntimeException("Error in select()");
            } else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer(abiList);
                peers.add(newPeer);
                fds.add(newPeer.getFileDescriptor());
            } else {
                boolean done;
                done = peers.get(index).runOnce();

                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
```

First, the file descriptor list is converted to an array and the index gets filled with a readable file descriptor. If that index is zero, a new `ZygoteConnection` is established which is listening on its defined socket and accepting pending connections. Afterwards, it gets added to the peer and `fds` list. The `acceptComandPeer()` method does call the `ZygoteConnection` constructor with `sServerSocket.accept()` as transfer parameter which is an extension to the `LocalSocket` implementation. Its `accept()` method accepts a new connection to the socket and is blocking processes until a new socket arrives. Therefore the next iteration delivers an index greater than

zero so that the last `else` branch of Listing 3.1 is executed. The `ZygoteConnection` object of `peers` located at that index calls the `runOnce()` method and gets removed out of the array lists afterwards. Finally, the `runOnce()` method will call `Zygote.forkAndSpecialize()` that forks a child within an exception which is being called to invoke the child's `main()`. But first, the given arguments from the command socket must be parsed with the aid of an `Arguments` class. Attributes that are needed for the later `fork()` call are shown in Table 3.6.

Table 3.6: Arguments Class Attributes

Given Argument	Attribute	Description
--setuid	int uid	UNIX uid for the child
--setgid	int gid	UNIX gid for the child
--setgroups	int gids[]	additional groups
--enable-debugger	int debugFlags	debug information
--enable-checkjni	int debugFlags	debug information
--enable-assert	int debugFlags	debug information
--enable-safemode	int debugFlags	debug information
--enable-jni-logging	int debugFlags	debug information
--mount-external	int mountExternal	storage to mount
--target-sdk-version	int targetSdkVersion	target version
--classpath	String classpath	absolute classpath
--runtime-init	boolean runtimeInit	new runtime init
--nice-name	String niceName	process renaming
--instruction-set	String instructionSet	instruction set to use
--seinfo	String seInfo	SELinux infos
--rlimit	ArrayList<int[]> rlimits	resource limitations
--app-data-dir	String appDataDir	data directory

Afterwards, all the defined security policies are being applied. To avoid bad file descriptor messages after forking a child, a native code has to close the open sockets before (`sServerSocket` and the local socket `mSocket` of the `ZygoteConnection` class whose FDs are written into an `fdsToClose` array). Now, all prerequisites for forking are fulfilled and the static method `forkAndSpecialize` of the `Zygote.java` file can be called (see Listing 3.2) returning the new process PID.

Listing 3.2: Zygote Fork Call

```
pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
          parsedArgs.gids, parsedArgs.debugFlags, rlimits,
          parsedArgs.mountExternal, parsedArgs.seInfo,
          parsedArgs.niceName, fdsToClose,
          parsedArgs.instructionSet,
          parsedArgs.appDataDir);
```

`Zygote.java` is quite compact since it mainly defines the transition to native code whereas the actual forking is being applied on Linux level. That is why a native version of the fork and specialize method are defined and executed which in turn returns the resulting PID after forking (`com_android_internal_os_Zygote.cpp`). The native implementation does finally call the effective `fork()` method that copies the actual Linux process. Afterwards, the child is being specialized (selected by the resulting PID of `fork()`). Transfered FDs are closed and capability boundaries are applied. UID and GID are being set (Linux `setresgid/setresuid`) and resource limits are added via `strlimit(2)`. If necessary, a native bridge will be initialized and scheduler policies are being set up. SELinux properties are applied and the thread name gets changed to a name other than “app_process”, usually the package name of the App.

Back in the Java world, `ZygoteConnection` checks the returned PID and either calls `handleChildProc()` in case of the child or `handleParentProc()` in case of the parent (Zygote itself). These methods are handling the post fork setup. In case of the child, sockets are being closed on Java level (since the child should not listen on sockets like the parent Zygote) and remaining arguments are being evaluated. When forking an App, a class name will be defined at the same time. Remaining arguments are copied to `mainArgs` and a class loader (`ClassLoader cloader`) is being defined before calling `invokeStaticMain()` of `ZygoteInit` with `cloader`, `className` and `mainArgs` as parameters. The class loader `cloader` is defined either with the `PathClassLoader()` or `ClassLoader()` constructor depending on if a classpath is given (path to APK or raw *.dex) where `PathClassLoader` is a specialized version of `ClassLoader`. However, when looking into the class loading code at this point, it becomes clear, that the DEX file of an App is used to represent a class and later finding its main method, so that it can be said for sure that an App’s DEX file is needed at least as an entry point under ART.

The `invokeStaticMain()` loads the specified class via the `className` string and furthermore searches for the main method, storing it as a Java `Method` object and finally throwing an `MethodAndArgsCaller()` exception. This exception gets caught by the main method of `ZygoteInit`. Remember that the Zygote itself is “trapped” in a loop whereas the child process can escape from it by throwing that exception. So as a result, Zygote will remain in the loop offering a socket to fork itself again (=starting a new App) while the child process (=the App to start) escapes from that

loop invoking the main method of the class specified over the socket connection. To clarify the program flow, Figure 3.5 displays the object and file interaction during the forking process.

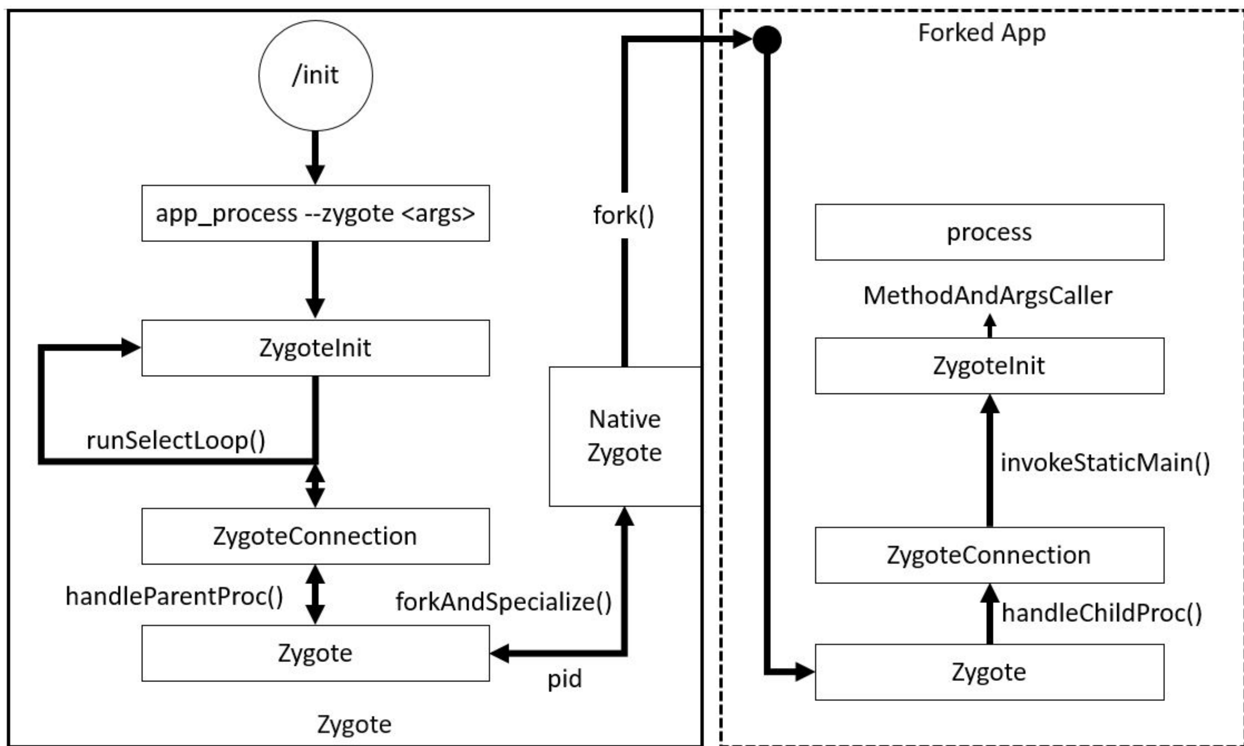


Figure 3.5: Zygote Forking

The current state of the child process is the one right after the main loop breakout of `ZygoteInit` induced by the exception. The exception is actually defined as a class that extends `Exception` and implements `Runnable`. Therefore it is viable to call the exception callers `run()` method (see Listing 3.3, taken out of the main method of Zygote).

Listing 3.3: Zygote Child Loop Breakout Exception

```
} catch (MethodAndArgsCaller caller) {
    caller.run();
} catch (RuntimeException ex) {
    Log.e(TAG, "Zygote died with exception", ex);
    closeServerSocket();
    throw ex;
}
```

Invocation means basically injecting the code of a method into the current process. That is the next and final thing that will be performed and it won't be explained in greater detail what happens.

Since most of the steps are performed in Java and class loaders are being used to represent the class and main method to be loaded, it is pretty clear that the DEX file is mandatory to locate the native code methods even at ART. If the reader is interested in checking the AOSP by himself,

Table 3.7 shows the explained source files and their paths inside of the AOSP branch of Android version 6 in this case.

Table 3.7: Zygote/App Start AOSP(6.0) Files

Path	Filename
/frameworks/base/cmds/app_process	app_main.cpp
/frameworks/base/core/java/com/android/internal/os/	ZygoteInit.java
/frameworks/base/core/java/com/android/internal/os/	ZygoteConnection.java
/frameworks/base/core/java/com/android/internal/os/	Zygote.java
/frameworks/base/core/jni/	com_..._os_Zygote.cpp
/frameworks/base/core/jni/	com_..._os_ZygoteInit.cpp

So what actually happens when a user clicks on an App icon? The Launcher App that is packaged with the AOSP takes care of the interface responsible for showing App icons, the home screen and so on. The App icon click will call the corresponding `onClick()` method that will in turn call `startActivity()` of the Activity Manager service through the Android Binder. The service will start `startViaZygote()` which opens a socket to the Zygote process requesting to start the new Activity/App [Yag13, ch.2, System Services].

3.2 DEX Disassembly and Repackaging

As explained in chapter 1 there are different goals of copy protection mechanisms starting from preventing reverse code engineering to protect intellectual property and reaching to hinder patching to get prohibited access. The common ground of those goals, is the protection of the DEX file of every App, since every distributed App includes it. A variety of tools exist that are able to transform DEX into different readable formats, modify it and repack it again since DEX contains a lot of meta data about its contents (classes, methods, . . .) [Goo16d]. Generally, there are two possible outcomes of DEX disassembling - Java code (`*.java`) and Smali code (`*.smali`). Since the DEX format is more or less just a different mapping of a JAR and its containing `.class` files, the transformation to JAR is quite simple [Bor08]. One of many tools that is able to perform this step is "`dex2jar`" [Pan16]. Along with this JAR, standard Java decompilers like "JD-GUI" [Unk16] can be used to produce the `*.java` source code. If the `*.java` is supposed to change and to be repacked, it can be compiled into JAR with Oracle's "`javac`" [Ora16] followed by Google's "`dx`" tool [Goo16a] to produce the new manipulated DEX.

An alternative way is the use of the “smali/baksmali” tool [Gru16] which is a direct assembler and disassembler for DEX files rather than taking the Java code detour. There is also a tool included that can convert the ODEX back to DEX (which is interesting for Dalvik Runtime systems).

Overall, the disassembly of unchanged DEX is quite easy and its main tools and possible conversions are shown as a concluding overview in Figure 3.6

Therefore, several countermeasures were established, which are described in the following sections.

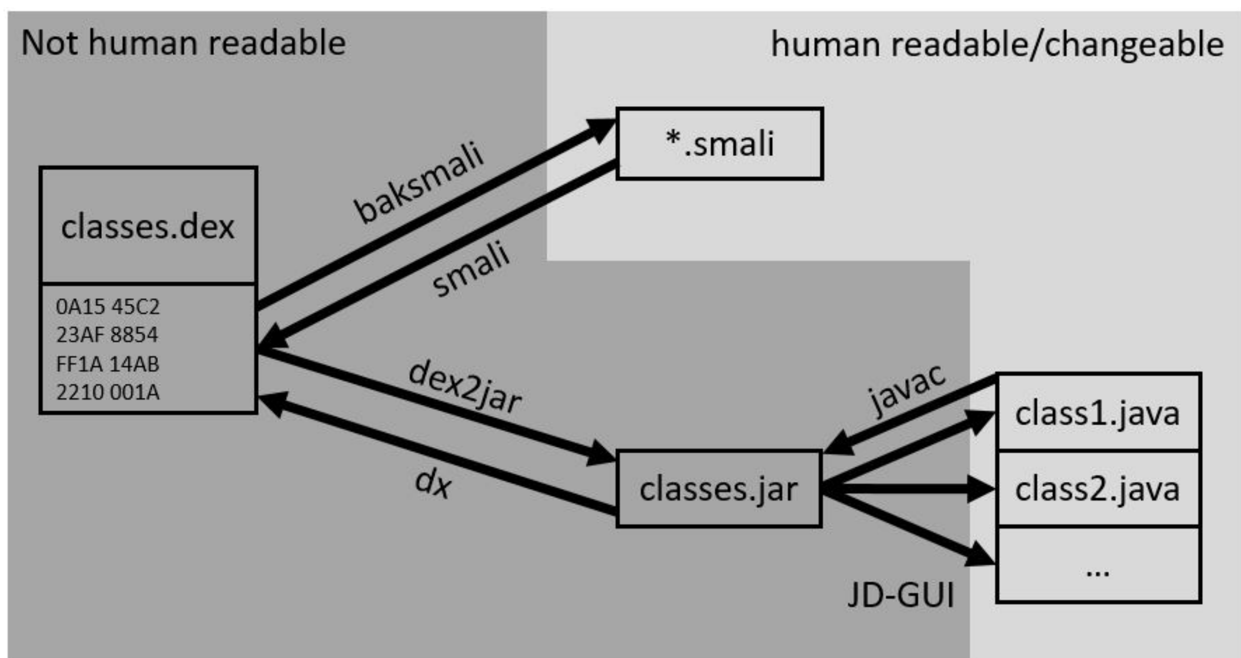


Figure 3.6: DEX Assembly/Disassembly

3.3 Obfuscation Techniques

Obfuscation in the context of copy protection for applications, is generally the term for hardening an application against reverse code engineering techniques. It can be achieved by different methods, that can be separated in two main groups, static and dynamic obfuscation. Static means, that the obfuscation technique is applied to code units (source code, binaries, ...), while the application is not being executed. Therefore, an attacker could possibly successfully analyze the application, without executing it, if he manages to break this obfuscation. An upside of static techniques in general is, that they are independent of the used runtime, if two compared runtimes are using the same input files which they do in case of Android with its DEX formatted files.

Applications that are dynamically obfuscated on the other hand, are much harder to analyze. The behavior of the application is not determined until its execution. An attacker needs to connect to the process of the running application followed by a just-in-time inspection. Dynamical methods have the downside that they are highly dependent on the runtime.

It follows a list of common static and dynamic obfuscation techniques for Android applications. However, this list is mainly focused on the Dalvik runtime since ART has been released quite recently. Where static obfuscation techniques show the same behavior in both runtimes, the dynamic solutions don't necessarily since the execution process of Apps in ART differs (described in subsection 3.1.2). The impact of those techniques to ART will get analyzed at every specific case.

3.3.1 Static

Common Source Code Obfuscation

The most common and simple way to harden source code is to remove any kind of meta data that has been added during the development process. This means destroying/modifying information that originally was present in the source code. Possible approaches of doing this are the renaming of string identifiers of classes, variables, methods and functions, artificially inserting irreducible code, creating artificial parallelization, performing method inlining/outlining, unrolling loops, encoding strings or changing the control flow in order to confuse code analysts by keeping the original behavior [Mun14, p.87].

Popular tools for that purpose are Google's "ProGuard" [Goo16f] which is included in the Android build system and can be enabled easily as well as "DexGuard" by GuardSquare [Gua16]. "ProGuard" does operate on source code level while "DexGuard" operates on DEX. Since the first "layer" of Android applications is Java code, classical Java obfuscators also can be used. Those tools do operate on the DEX file layer, meaning that they can be applied at ART without restrictions.

Junk-Byte-Insertion

Junk-Byte-Insertion's goal is to prohibit the use of program analyzing disassembling tools. It works for tools using the "linear sweep" method to analyze a program. That means the tools are processing every instruction from the entry-point till the end without interpreting them (e.g. not following jumps). That examining technique can be exploited to break the disassembling

procedure. Let's assume we do have the DVM instructions shown in Listing 3.4(Example taken out of [Mun14, p.67]).

Listing 3.4: Junk-Byte-Insertion

```
if "true" goto line 4
load_array_into v1, line_3
array_size 10
set v2, 1
set v3, 1
add v1, v2, v4
return v4
```

Because of the if-statement that performs a jump to line four, the `array_size 10` command will never be reached. Since "linear sweep" does not perform jumps, the analyzing tool will interpret the ten following bytes of that array initialization as payload and will therefore not be able to disassemble the actual instructions.

Enhanced tools will use the "recursive traversal" technique to analyze a program, which is capable of detecting dead branches and conditional jumps like in the example above. These tools also may be tricked by choosing a more complicated condition in the if-conditions, that can only be evaluated at runtime. This would result in the tool trying to evaluate the whole conditional branch (including the breaking byte sequence). Actually, this technique could already be counted to dynamic obfuscation [Mun14, p.68].

This example was based on DVM instructions and was therefore designed for Dalvik/Android prior version five. The Junk-Byte Insertion will no more be possible in ART. Why? Even if one would insert Junk-Bytes into the DEX file, there is still a conversion step into the new OAT format applied in ART, which compiles into native code so that none of the inserted bytes will be adopted. In summary, Junk-Bytes can still be inserted into DEX to prevent an attacker from disassembling of DEX but those Junk-Bytes will not apply to the compiled code. However, since the compiled code is native code and therefore not that easy to disassemble, Junk-Bytes might still be usefull at DEX layer.

3.3.2 Dynamic

Hidden Methods Invocation

In [Mun14, p.82f], a technique is described to hide a whole method in `.dex` files. This hiding method is highly dependent on the DEX specification from Google [Goo16d] that has been described in section 3.1.1. It is based on the fact that the actual instructions of methods residing in the `data` section of a DEX are referenced but not parsed directly by sweeping over that file area, so that the meta data section of methods includes an offset to its actual instructions. These

references (which are offsets in the data section), can be manipulated to hide specific methods when the DEX gets parsed. Like shown in Figure 3.7, one could manipulate an offset pointer (in this example the offset from method one) to hide its implementation. In order to achieve this effect, the offset value needs to be changed. Since the DEX format includes a checksum to be resistant against transmission errors, a revaluation is necessary. After that hiding step, the method is invisible for static analyzing tools but of course also for the DVM itself, meaning it can't be run anymore. Thats why the changes to the DEX need to be reversed at runtime.

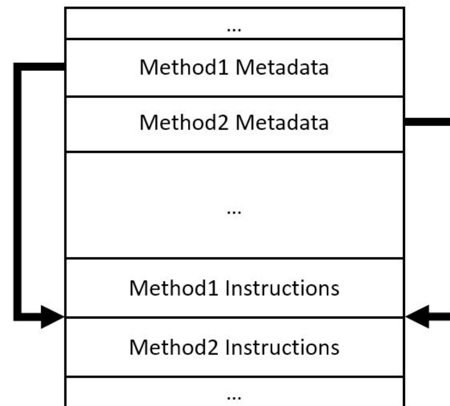


Figure 3.7: Hidden Methods Invocation Principle (Method1 gets hidden)

A precondition for the Hidden Methods Invocation technique is the possibility to load DEX files dynamically at runtime in order to revert those changes. Otherwise a manipulation of the DVM bytes in a running App in RAM would be necessary to revert them.

The next section deals with dynamic code loading in general and will therefore cover the feasibility for Dalvik and ART.

Dynamic Code Loading

The principle of Dynamic Code Loading is to reveal the actual program code only after running the application. This behavior can be achieved by implementing a stub code that will then load the actual application followed by its execution. The format of that distributed application is in DEX format, at least in Dalvik. If DEX still can be used in combination with ART, has to be determined. In [Goo16e] a practical sample implementation for Dalvik is given. To provide a copy protection benefit, an additional application file can either be distributed encrypted within the App stored in the `assets` folder, or it can be fetched from a server. Android provides a public method within the `DexFile` class to dynamically load DEX files (`loadDex(String sourceName, String outputName, int flags)`) and loading included classes. The `sourceName` parameter accepts a path to a JAR or

an APK and `outputPathName` specifies the file in which the optimized DEX version will be saved (ODEX in Dalvik). That behavior however is a problem, because the created ODEX will still persist also after closing the App and therefore can be analyzed after a one time execution. Unfortunately it also can't be deleted because of missing writing permissions. In [Sch12], a circumvention for that problem is described by using the JNI. The `libdvm.so` library offers a private method to open DEX files that accepts DEX content in form of a byte array. By establishing this own JNI implementation of `openDexFile(byte[]content)` the loaded dex file is only present in volatile memory and does not create an ODEX [Sch12] which makes that method very robust. So the big question here is if that method also can be applied to ART since `libdvm` has been replaced with `libart`.

First it will be tried to load a DEX in ART with given Java APIs. As mentioned, `loadDex()` is a possible method but should be called through a `DexClassLoader()`. The method expects a `dexPath` String which defines where to look for DEX files, an `optimizedDirectory` String specifying a path to store the optimized version as well as a `parent` ClassLoader. The DEX to being load can be shipped within the assets folder. So to begin with, a DEX is needed. For simplicity reasons a simple Java class will be used, shown in Listing 3.5.

Listing 3.5: Java Class to load

```
public class ToLoad{
    public int exampleMulMethod(int a, int b){
        return a * b;
    }
}
```

A DEX can be created with a "`javac ToLoad.java`" call followed by using the `dx` tool with "`dx --dex --output="toload.dex" ToLoad.class`". Be sure to use a Java version < 1.8, otherwise the `dx` command will fail since Java 1.8 is not yet officially supported by Android at the time of writing this thesis. The `dexPath` has to be readable by the App itself. Therefore, the DEX should be stored in the private App data directory where dexPath should then search for it. Same goes for the optimized directory. `getDir()` can create a private folder with a chosen name and privileges. With `getAssets().open()`, the DEX can be copied out of the assets folder to the new data location by using buffered Java streams (see Listing 3.6).

Listing 3.6: Dex Internal Storage

```
File dexInternalStoragePath = new File(getDir("dex", Context.MODE_PRIVATE), "
    toload.dex");
BufferedInputStream bis = null;
OutputStream dexWriter = null;
final int BUF_SIZE = 8 * 1024;
try {
  bis = new BufferedInputStream(getAssets().open("toload.dex"));
  dexWriter = new BufferedOutputStream(new FileOutputStream(
      dexInternalStoragePath));
  byte[] buf = new byte[BUF_SIZE];
  int len;
  while ((len = bis.read(buf, 0, BUF_SIZE)) > 0){
      dexWriter.write(buf, 0, len);
  }
  dexWriter.close();
  bis.close();
}catch (IOException e){
  e.printStackTrace();
}
```

Now, the `DexClassLoader()` constructor can be called with the recently created paths and the context class loader. After that, the class can be loaded using its name and the method can be received together with defining its argument classes. Finally, `invoke()` can be called with a reference of the object of being called from (`null` if method is static, `newInstance()` if not) and its arguments (Listing 3.7).

Listing 3.7: Dex Method Invocation

```
final File optimizedDexOutputPath = getDir("outdex", Context.MODE_PRIVATE);
DexClassLoader dcl = new DexClassLoader(dexInternalStoragePath.getAbsolutePath
    (), optimizedDexOutputPath.getAbsolutePath(), null, getClassLoader());
Class classToLoad = null;
Method m;
try {
  classToLoad = dcl.loadClass("ToLoad");
  m = classToLoad.getDeclaredMethod("exampleMulMethod", int.class, int.class);
  TextView textView = (TextView) findViewById(R.id.invokeResult);
  textView.setText("8*9=" + m.invoke(classToLoad.newInstance(), 8, 9));
} catch ...
```

The optimized output file is in OAT format. This way, the dynamic DEX loading is generally possible in ART with purely official Android Java methods. However, the execution of code loaded that way should be very slow since the DEX file is optimized in an ART way (=compilation into native code) and it doesn't really fit into the ART AOT compilation philosophy. Of course, when using this method to initialize an App right after installation, it still might be useful for copy protection applications. With that in mind, it should be clear that it is not possible to load a DEX file into a byte array and execute it directly like it was possible at Dalvik using the "`libdvm`" library via JNI. The reason should be clear, since ART does not operate on DVM byte-code and therefore cannot execute those DVM instructions. So in the scope of copy protection mechanisms

with ART, dynamic code loading at Java layer is not that useful anymore but can still be used to dynamically load App parts in general. Nevertheless it needs to be analyzed how code can be loaded dynamically in ART which will be explained in chapter 4. To come to the point right away: Dynamically loading code is possible, mainly by using native code and therefore JNI functionalities.

Self Modifying Code

Quite similar to dynamic code loading described in section 3.3.2 is self modifying code with the goal of altering the instructions of an App during runtime. Instead of loading additional code snippets, the focus lies on manipulating the executing Dalvik byte code stream directly. The DVM is limited in terms of instructions for modifying byte code and therefore the byte code world has to be bypassed with native code using the JNI. To find the position in the byte code that should be changed, a predefined value must be set in order to be recognized by the native code function. That value is often called “egg” and the search process “egg-hunting”. Let’s assume we have the code snippet shown in Listing 3.8 that exists in the context of an Android App activity [Sch12].

Listing 3.8: Self Modifying Code Example

```
...
modifyVariable();
int egg = 0x12345678;
Integer toChange = 5;
...
native private void modifyVariable();
...
```

The `modifyVariable()` is a native code method defined over the JNI and sweeps over the process memory (detectable by evaluating `/proc/self/maps`) in order to find the egg value. After skipping the assignment of the egg value, the next instruction is responsible for allocating the `toChange` variable. In this case, it is “`0x13 0x21`” and does stand for the mnemonic “`const/16 vAA, #+BBBB`” [Goo16c]. Therefore the next byte specifies the register that needs to be saved followed by two bytes of the signed integer value (“`0x05 0x00`” in our case). By changing the “`0x05`” to “`0x09`” the goal of dynamically changing the value at runtime is fulfilled.

This example was again for DVM instructions and is described in [Sch12]. So an interesting question is if code manipulation on the fly is still possible at ART. It will get analyzed in great detail in the next chapter. What should be clear by now is that the dynamic code loading or manipulation on the fly can only be done in native code so that there might be a way to dynamically load whole code snippets.

Chapter 4

Android Dynamic Native Code

This chapter will shortly cover how to write native code in Android, how the connection between Java and C/C++ is being established as well as the native code integration into Android Studio. Then there will be a sample application that shows a dynamic native code execution of a native code shared library, a binary or machine code that can either be shipped within `assets/` or can be pulled in from external sources like the web, Secure Elements (SE) and so on. It will also be covered if and to what extent an Android App has access to its own memory location as well as to its mapped files considering its writing and execution permissions. With C/C++ there should be multifarious possibilities to do that if sandboxing/permission concept mechanisms do play along.

4.1 NDK and Android Studio Integration

A complete documentation and starting guide about the NDK can be found at [Goo16b] and a guide to the integration into Android Studio at [Glo15]. There is more than one way to integrate the NDK into Android Studio. The one described in this thesis uses the platform build tools script `ndk-build`. Another possible integration is the use of the Gradle experimental plugin and is described in [too15]. The NDK allows to embed compiled C/C++ in an application. It can be installed via Android Studio which is the default and recommended IDE for Android development. The result of compiled C/C++ code is CPU dependent. Therefore the library has to be cross compiled for every possible CPU supported by Android. Currently possible architectures supported by the NDK are:

- **arm64-v8a**
- **armeabi**
- **armeabi-v7a**

- **mips**
- **mips64**
- **x86**
- **x86_64**

The common way is that Android compiles every native source code into shared libraries (`*.so`) that can then be loaded from Java code. Theoretically, it is also possible to build executable binaries with NDK. Entry point for the build process is an `ndk-build` script inside of the installed `ndk-bundle/` folder that takes care of calling the cross-compiling toolchains. It can be controlled by Android makefiles (`Android.mk`) and requires a specific folder structure to work which is partially created when using Android Studio. An additional `jni/` folder has to be created that will hold all the native source code (right click on `app/` and choose `New->Folder->JNI Folder`).

The compiled library can be loaded from Java by using a `System.loadLibrary("MyLib")` call. In order to be able to call methods out of that library, the method signatures have to be known. They can be declared with a `native` keyword in addition to the common Java method declaration. It is also a common practice to create a new Java class specifying all NDK functions but it is not a must have. When doing so, a header file for the native source code (`*.h`) can be automatically created by using the `javah` command line tool. It includes the necessary JNI declarations for the specified methods (including the JNIEnv pointer for instance) and can then be included in the C/C++ source file that can afterwards be implemented as a common C/C++ project.

The last necessary step is to create an `Android.mk` makefile for every library as well as an`Application.mk`. The `Android.mk` file defines the library name that `loadLibrary()` expects, lists the source files and defines the outcome (shared library or executable). `Application.mk` includes the libraries to build (`APP_MODULES`) and the architectures to build for (`APP_ABI`). The `build.gradle` has to be adapted as well, defining the outcome directory path relative to `jni/` and `android.useDeprecatedNdk = true` needs to be added to `gradle.properties` to suppress an error message. A `ndk-build` call will then compile all sources and creates the libraries at `libs/<archs>`. They can be used out of the box without need to load them by hand at runtime. All that's needed is the `loadLibrary()` call. Physically those libraries are stored at the same path as the App APK file as introduced earlier. By applying this method, those native libs to load are shipped together within the application that might not be the intended goal. A whole NDK sample project including the explained adaptations can be found in Appendix A.

4.2 Dynamic Shared Library Loading out of a File

The process demonstrated in section 4.1 is common practice to integrate native code into an App but is not really dynamically loaded. Instead the main idea of dynamical code loading is the distribution of code after licensing the App or to apply some sort of copy protection mechanisms by hiding the true functionality. One approach is to use native code as a loading mechanism. C/C++ has the ability to load shared objects (*.so) with a method called `dlopen()` (="Dynamic Library Open"). Its signature is shown in Listing 4.1.

Listing 4.1: dlopen() Signature

```
void *dlopen(const char *path, int flag)
```

In order to enable dynamical loading, a path to the library file is needed as well as a flag that defines the binding of variables and methods (lazy, global, ... see `dlopen()`-reference for more information). But what path should and can be used in the context of Apps? A path inside of the APK for example is not referenceable as a path string. So generally, Apps can use different storage options that are described below. They can also be looked up in [Goo16g].

- **Shared Preferences** can store data in form of key-value pairs
- **Internal Storage** stores private arbitrary data on the device memory at the path `/data/data/<appPackage>/` that is only accessible by the App itself.
- **External Storage** can also store arbitrary data on `/sdcard/` but it is not exclusively readable by the initializing App and needs a permission declaration in the manifest file.
- **SQLite Database** provides support for powerful private databases
- **Network Connection** can store data on the web

The most suitable option should be the internal storage since the App has unlimited read/write rights and the content is safe from other non-root applications. For simplicity and demonstration purposes, the library to load will be stored in the `assets/` folder of the App that also only exists in the APK. It is not referenceable through a path but can be opened through Asset-Manager. It will therefore be copied into the internal storage first. Listing 4.2 shows the Java initialization of the path to the final library to load. The `getDir()` call generates a new folder within the internal App storage and a file container for the library to load gets created.

Listing 4.2: Internal Storage Initialization

```
File internalStoragePath = new File(getDir("dynamic", Context.MODE_PRIVATE), "
    mul.so");
```

As a next step, the library has to be copied out of the assets folder in the container which has been created for it in a previous step. Any file within the assets can be opened with an `getAssets().open("file")` call. For the read/write process one can use `BufferedInputStream` and `BufferedOutputStreams`, shown in Listing 4.3.

Listing 4.3: Buffered Input/Output

```
BufferedInputStream bis = null;
OutputStream soWriter = null;
final int BUF_SIZE = 8 * 1024;
try {
    bis = new BufferedInputStream(getAssets().open("mul.so"));
    sWrite = new BufferedOutputStream(new FileOutputStream(internalStoragePath)
        );
    byte [] buf = new byte[BUF_SIZE];
    int len;
    while ((len = bis.read(buf, 0, BUF_SIZE)) > 0){
        sWrite.write(buf, 0, len);
    }
    sWrite.close();
    bis.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Now, the program is ready to transform the transition into the native (C/C++) world. The signature of the native method to call contains at least the path to the internal stored file as a string that can be printed with a `getAbsolutePath()` call via the `internalStoragePath` file object. Remember, that `dlopen()` needs that path to load the actual library. A class "`MyNDK`" is created and contains the native declaration of the "`libExe(String path)`" method as well as a static call of `System.loadLibrary(‘‘MyLib’’)` that takes care of the binding. The implementation of `libExe()` is shown in Listing 4.4.

Listing 4.4: Native libExe()

```
JNIEXPORT void JNICALL Java_ma_schleemilch_nativestuff_MyNDK_libExe
        (JNIEnv * env, jobject jobj, jstring path){
        const char *libpath = env->GetStringUTFChars(path, NULL);
        LOGD("Received␣Path:␣%s", libpath);
        void* handle;
        const char* error;
        long (*mul)(int, int);

        handle = dlopen(libpath, RTLD_LAZY);
        if (!handle) {
                LOGE("DL␣Open␣failed:␣%s", dlerror());
                return;
        }
        dlerror();
        *(void**)(&mul) = dlsym(handle, "mul");
        if ((error = dlerror())!= NULL) {
                LOGE("DL␣Error␣after␣DLSYM:␣%s", error);
                return;
        }
        LOGD("#␣9*5␣=␣%ld", (*mul)(9,5));
        dlclose(handle);
        remove(libpath);
}
```

A short remark about logging/debugging: The common `printf()` writes to the standard output. A better way of viewing C++ outputs is to use the ADB logging mechanism. Google includes a `__android_log_print()` function that has the same behavior as `printf()` but writes to the Logcat output where `LOGD()` is a defined makro calling that function.

The library that will get loaded is a simple multiplication function and a pointer for it is initialized at line seven. `dlopen()` returns a void pointer and `dlsym()` searches the symbol table of the handle pointed file for the function handed over as a string. That call is interesting when comparing C to C++. In [Iso06], the problem of name mangling in C++ and the combined usage of `dlopen()` is described in detail. In short, one will need an "`extern "C"`" at the `mul()` definition in order to enable the `dlsym()` function to find the method because C++ does not only use the method name as its symbol but additionally a unique created ID. So without that `extern` keyword, `dlsym()` will fail. Finally, the `mul()` function can be called from the library. After that call, `dlclose()` has to be executed. It is also possible to remove the library file after that process which might be useful for copy protection usage. From the Java perspective, a `MyNDK` object needs to be created followed by a call of its `libExe()` to call and execute the library.

To be able to load a shared object library like this one, it obviously needs to be generated first. One could use an own cross compiler to do so or just use the NDK since it includes all necessary toolchains. However, some project structure changes have to be made to be able to compile two independent libraries. Each library to compile should have its own folder inside of `jni/` with an `Android.mk` each. The greater `Android.mk` in `jni/` has to be changed to call all subsequent

make files with "`include $(call all-subdir-makefiles)`" and each module to build has to be added to `Application.mk`. Gradle needs to know the names of the sub- directories of `jni/` so that "`jni.srcDirs=[]`" has to be filled with those folder name strings. As a result of that structure, the compiled library that will be loaded afterwards also will be saved in the `libs/` folder and shipped within the App if not deleted beforehand. For instance, in a scenario where an App requests new parts and gets them from a server, these libraries should only be present on the server side. In order to receive the right compiled shared object for every device, the request needs to contain CPU information (e.g. by sending the output of "`System.getProperty("os.arch")`").

It is also possible to do that kind of shared object loading directly out of the Java code. Like shown in section 4.1, a static call of `System.loadLibrary("MyLib")` does the heavy lifting of invoking the library. The `loadLibrary()` function however does only check specific paths for the given library name which are specified in `java.library.path`, containing `/vendor/lib` and `/system/lib` as well as the path of the App's APK lib folder. So for dynamic loading, this loading method is not suitable since these locations can't be accessed without root rights. Fortunately there is another loading function `System.load(String s)` that accepts an absolute path to the shared object file like the C++ equivalent `dlopen()`. Attention must be paid to naming conventions in `loadLibrary()` in comparison to `load()`. If libraries are compiled by the NDK a `lib` prefix is automatically added to the module name. In short, it means that `System.loadLibrary("MyLib")` invokes a file that's called `libMyLib.so` while the `load()` requires the physical stored path name.

Even though Java is also capable of loading shared object libraries at runtime, the signatures of the library to load have to be implemented beforehand to be able to call library methods. This is a different procedure compared to C/C++, where symbols can be resolved dynamically.

It can also be tried to call a whole native activity with the C/C++ method since a native activity project can be compiled into an shared object (see NDK examples "native-activity"). To invoke the activity, one needs to call the `android_main()` method but when doing so, but this will cause a segmentation fault (SIGSEV).

4.3 Dynamic Binary Execution out of a File

It could be also interesting to execute binaries directly instead of loading shared objects. To compile them, the NDK can be used again but the module's `Android.mk` file needs to be adapted by replacing `include $(BUILD_SHARED_LIBRARY)` with `include $(BUILD_EXECUTABLE)`. When doing so, it needs to be taken into account that all modules compiled for 32 Bit systems are directly executables while modules compiled for 64 Bit are shared objects (by checking the output of

the Linux `file` command line program). Either way they are dedicated to their interpreters (`/system/bin/linker` and `/system/bin/linker64`) that will execute them. It does however make a difference since Android seems like to only support position independent executables (PIE). PIE's purpose is to be able to execute its code regardless of its absolute address. If the reader is interested in more information about PIEs, [Mur12] is an good article about the impact of PIEs. Shared objects are a specific implementation of PIEs.

In case of 32 Bit systems, there will be an error message "`error: only position independent executables (PIE) are supported.`" when compiled as executable without any additions. A solution for this issue is adding a `LOCAL_CFLAGS` entry `-fPIE` as well as the `LOCAL_LDFLAGS` entries `-fPIE` and `-pie` to the Android makefile. In fact, after adding those attributes, there is no difference between the compiled executable and the shared library counterpart anymore other than it has to contain a `main()` in order to be executed.

Again, binary execution can be implemented in Java as well as in native C/C++ code. Java will be used in both cases to fetch the binary file from an arbitrary location that can be determined at runtime or statically (again `assets/` for demonstration) and to write it into the internal private storage path. When checking the outcome file of that process via `adb shell` and "`ls -l`", the file is by default not marked as executable but at least the owner and the group is set right so there should not be any permission issues. If the file is not marked as an executable, it will result in a "can't execute: Permission denied" message in case of a C++ implementation or an "IOException" in case of a Java implementation. Fortunately, this can be fixed by simply calling `setExecutable(true)` of the Java `File` object which represents the binary.

4.3.1 Java Implementation

The preparation procedure for executing is the same as in the shared object loading case, which means the file is saved in the App's internal storage and a `File` object to work with is delivered. A Java `Runtime` object has an `exec(String s)` function that is capable of executing code given its absolute path. It returns a `Process` object. In Android, the current runtime can be accessed with a static access of the `Runtime`'s `getRuntime()` function. That is basically everything needed to make it work. However, there are not any outputs (like `printf()`) of that separately started process. The documentation of `Process` reveals that the subprocess output can be fetched with `getInputStream()` and its error output with `getErrorStream()`. The `waitFor()` function can be used to check and wait for the subprocess to finish which should be done when calling a native binary. The final, very compact implementation is shown in Listing 4.5.

Listing 4.5: Java Native Exec()

```
Process nativeExe = Runtime.getRuntime().exec(internalPath);
BufferedReader reader = new BufferedReader(new InputStreamReader(nativeExe.
    getInputStream()));
int read;
char[] buffer = new char[4096];
StringBuffer output = new StringBuffer();
while ((read = reader.read(buffer)) > 0) {
    output.append(buffer, 0, read);
}
reader.close();
// Waits for the command to finish.
nativeExe.waitFor();
String nativeOutput = output.toString();
Log.d(TAG, "nativeOut: " + nativeOutput);
```

4.3.2 C/C++ Implementation

As for C/C++, the implementation is not much more complicated compared to the Java implementation. The main part of the execution is a calling `popen()` (man printed at [Ker15]) which requires a command given as a char array and the type of its created pipe (read "r" or write "w"). Internally, it uses `fork()` and `pipe()` for creating the new process and executing the given program at its path. It returns a `FILE` pointer whose stream can be read out using `fgets()`. The complete C++ implementation of the behavior is shown in Listing 4.6. In order to catch `stderr`, the program path string can be extended with the string "2>&1" to detour the error channel (2) and route it to the common output channel `stdout` (1).

Listing 4.6: C++ Native Exec()

```
JNIEXPORT void JNICALL Java_ma_schleemilch_nativestuff_MyNDK_binExe (JNIEnv *
    env, object obj, jstring path)
{
        const char *exepath = env->GetStringUTFChars(path, NULL);
        FILE* fpipe;
        char* command = new char[strlen(exepath) + strlen(" 2>&1") + 1];
        int ind = 0;
        for (int i = 0; i < strlen(exepath); i++){
                command[ind] = exepath[i];
                ind++;
        }
        command[ind++] = ' ';
        command[ind++] = '2';
        command[ind++] = '>';
        command[ind++] = '&';
        command[ind++] = '1';
        command[ind++] = '\0';
        char line[256];
        if (!(fpipe = (FILE*)popen(command, "r"))) return;
        while(fgets(line, sizeof(line), fpipe)){
                LOGD("%s", line);
        }
}
```

4.4 Dynamic Code Execution from Memory

Until now, dynamic code execution or loading via shared libraries was done by downloading/-copying a file into the internal storage of the App followed by its invocation. This is a detour since that file has to be fetched and written to a local file followed by loading it again. Instead, it would be also interesting to know if program code can be executed directly out of memory. Memory in this context means the RAM section of the App's process. So it needs to be inspected if an Android App process has even access to memory and if it is feasible to read/write/execute program code in self allocated memory.

4.4.1 Android Memory Mapping

Let's have a look at the general memory mapping of a common Android App. When printing the content of `/proc/<PID>/maps` of the App's Linux process, virtual memory locations that are assigned to the process are revealed. The keyword "`self`" is the specific PID that can be used as a path in a process to access its own `proc/` directory. Otherwise, root rights are needed to read maps of processes other than its own. An example output of a map entry is shown in Table 4.1 [Con09]. The `address` shows the absolute physical start and end of its mapped section. Permissions (`perms`)

Table 4.1: Content of /proc/<PID>/maps

address	perms	offset	dev	inode	pathname
6feed000-708ca000	rw-p	00000000	b3:1c	105876	/data/.../framework@boot.art

contains information about how this region can be accessed (common Linux "`rwx`" permission triple) and can be marked as private `p` (exclusively accessible by its mapped process) or shared `s` (accessible from other processes). If the section was mapped from a file, `offset` describes the offset in bytes to the specific region of the mapped file. Device (`dev`) only applies if mapped from a file and includes the device number while `inode` holds the mapped file number. The `pathname` shows the path to its source file.

When looking at an Android App process in particular, it reveals all kinds of mapped libraries like fonts, its own `base.apk` as well as regions that don't have a pathname or just a pseudo pathname like "`[anon:linker_alloc]`" or "`[stack]`". A great amount of libraries are mapped this way and are not using an own App specific copy of those libraries. Only if they are being used they will be copied. That behavior emphasizes that forking Zygote is way more efficient rather than copying all those libraries every time an App starts.

Generally, a stack contains data that only persist inside of a function call for example initialized arrays or variables. Their content are getting released automatically after returning from that function. The heap however can be used by the programmer to allocate memory manually for example with `malloc()` but has to be freed up explicitly. Let's evaluate which mapping sections are used to store heap as well as common stack elements by creating a native function with a buffer using `malloc()` and a common local integer variable. While `malloc()` creates raw memory areas without any additional flags, `mmap()` can map a whole file, returns a pointer to its start address and can set memory protection flags of the mapped area that are necessary when it shall be executed, written or read later in the process (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`). Executing code out of the heap is forbidden by default. Listing 4.7 shows how to read and print the mapped content from C++ on Logcat.

Listing 4.7: Reading /proc/self/maps

```
FILE* fp;
char line[2048];
fp = fopen("/proc/self/maps", "r");
if (fp == NULL){
    LOGE("Could␣not␣open␣/proc/self/maps");
    return;
}
LOGD("Before:\n");
while (fgets(line, 2048, fp) != NULL) {
    if (strstr(line,"triggeredString")){
        LOGD("%s", line);
    }
}
fp->_close;
```

Unfortunately, the amount of output lines is limited when using logging via Logcat which implements a ring buffer with a device dependent size ("`adb logcat -g`"). That is a problem when printing the whole memory mapping and can make an inspection confusing since a ring buffer overwrites itself when overflowing. A possible solution would be to use an adb shell executing "`cat /proc/<PID>/maps`" by hand (root rights needed) or to trigger the Logcat output for specific strings. Getting an overview using the whole output and triggering it for specific parts via Logcat afterward might be the best solution. To find the mapping of the executing library, triggering the name library is enough. Its output is shown in the top three lines of Table 4.2. Three regions of the library are mapped, with reading, writing and executing respectively. When creating an integer variable and printing its address afterwards, it appears that it is located in a "`[stack]`" marked region that is not included in the mapped ranges of the library and memory allocations by the `malloc()` as well as the `mmap()` function are listed in areas marked as "`[anon:libc_malloc]`" (printed in Table 4.2).

Table 4.2: Memory Allocation Mapping

address	perms	offset	dev	inode	pathname
b39f5000-b39f8000	r-xp	00000000	b3:1c	187593	.../lib/arm/libMemory.so
b39f8000-b39f9000	r-p	00000000	b3:1c	187593	.../lib/arm/libMemory.so
b39f9000-b39fa000	rw-p	00000000	b3:1c	187593	.../lib/arm/libMemory.so
be0c3000-be8c2000	rw-p	00000000	00:00	0	[stack]
b3940000-b39c0000	rw-p	00000000	00:00	0	[anon:libc_malloc]

4.4.2 Code Execution

So which function (malloc or mmap) is more suitable for just-in-time (JIT) code execution? As explained in [Ben13], one should go with `mmap()` over `malloc()`. This is because of protection bits that can only be set at memory page boundaries. The reason behind that is that with `malloc()` it needs to be ensured manually that the allocation is aligned at a page boundary. If this is not the case, a `mprotect()` call will have the side effect of disabling/enabling more memory than actually required. Since `mmap()` is designed for mapping whole files, it will take care of that issue by default.

In the end, “code” like in “Executing Code” stands for machine code that will get executed, which isn’t anything more than a sequence of bytes (actually only a 1 and 0 stream and no partitioned bytes). Up to this point, dynamic loading was performed through files that do have a specific format that contains far more than just raw bytes that are getting executed (like shared objects or ELF binaries, partially described in section 3.1.1) and API calls. There is a reason for those kinds of formats to be used since they offer information to the linker which enables them to function properly on every system using them. Generally there are two possibilities of invoking and executing code and therefore native instructions. The first one is to map a well known format like a shared object or an executable into memory but this would mean that the running “host” process needs to be adapted in order to find those executable parts in the memory. The linker is normally responsible for doing that before the program is executed. If a program is using external libraries, a linker is responsible for linking them into the main binary, statically or dynamically. So one would end up needing to write linker functionalities or patching the one mapped into the process. It exists a proof of concept for a Linux x86_64 system shown in [Mim15] that can load shared objects out of a file or an Internet socket right into the process space. A key part of this program is the patching of “`ld-2.19.so`” that is already mapped in the process. Porting that

library to Android is not that simple since Android is not using this library but an own linker at /system/bin/linker. It would have to be investigated how that linker can be called/adapted in order to achieve a similar behavior.

So let's first try to invoke machine code directly without using any file format overhead like in ELF binaries and shared objects. Again, a simple multiplication function can be implemented to show a proof of concept, shown in Listing 4.8.

Listing 4.8: machineCodeMul.c

```
int mul(int a, int b){
    return a*b;
}
```

This time, a cross-compiler and its included tools like objdump are useful. For this explicit purpose, the arm-none-eabi toolchain was used in combination with an Arch Linux 64 Bit system. It contains a gcc to compile C-Code as well as objdump to analyze object files. A "arm-none-eabi-gcc -O3 -c machineCodeMul.c -o machineCodeMul.o" call outputs an object file with a C source input. Objdump can then show mnemonics as well as its corresponding machine code for every function which will be written to memory in the next step (output for machineCodeMul.o's mul() function shown in Table 4.3). The byte sequence 0xe0000091 represents

Table 4.3: arm-none-eabi-objdump -D machineCodeMul.o

00000000 <mul>:			
0:	e0000091	mul	r0, r1, r0
4:	e12fff1e	bx	lr

the raw machine code for the mnemonic equivalent of a multiplication (r0 = r1 * r0) and the following sequence is responsible for exiting the function and returning the value. Since object files are normally Little Endian, the bytes written to memory have to be reordered (0xe0000091 turns to 0x910000e0) before execution.

At first, a memory location will be needed and allocated. For that purpose a function is written which takes the regions size as an input as shown in Listing 4.9.

Listing 4.9: alloc_executable_memory()

```
void* alloc_executable_memory(size_t size) {
    void* ptr = mmap(0, size, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE |
         MAP_ANONYMOUS, -1, 0);
    if (ptr == (void*)-1) {
        LOGE("mmap");
        return NULL;
    }
    return ptr;
}
```

It returns a void pointer to the starting address of the allocated region. Note that if no file is being specified as a `mmap()` parameter (instead a "-1"), the `MAP_ANONYMOUS` flag has to be set. Additionally, even though size can be specified in `mmap()`, it is limited to page size constraints. This way it will allocate memory that is greater or equal to the given size. Also to mention is that there might be a security hole since that allocated memory is already marked as writable and also that region is likely to be bigger than the actual code that will get executed later. This situation could be exploited by an attacker. Therefore it is slightly more secure to allocate memory without execution permission at first since copying machine code to that location does only need write permissions. The execution permission can be granted right before the function is called via `mprotect()`.

The next thing to do is to copy the bytes to be executed to that location. In order to do so, a function is used that takes a pointer, specifying some bytes and finally calling `memcpy()` (see Listing 4.10).

Listing 4.10: emit_code_into_memory()

```
void emit_code_into_memory(unsigned char* m) {
    unsigned char code[] = {
        0x91, 0x00, 0x00, 0xe0, // mul r0,r1,r0
        0x1e, 0xff, 0x2f, 0xe1, // bx lr
    };
    memcpy(m, code, sizeof(code));
}
```

But how can this code actually be executed? A `typedef` is used to specify the function's signature matching the C source function that will get executed afterwards. It can be initialized by pointing to the allocated memory region the way it does when writing a function the common way. The function name is then a pointer to the process region, including the actual machine code. After that, all that's left to do is calling it like a common function. Listing 4.11 shows the actual implementation via JNI C++ using the predefined methods from before and showing some additional output information about the allocated memory spots.

Listing 4.11: executeMachineCode()

```
JNIEXPORT void JNICALL
    Java_schleemilch_ma_nativememory_MyNDK_executeMachineCode (JNIEnv *env,
    jobject obj){
    typedef int (*JittedFunc)(int, int);
    size_t SIZE = 8;

    void* m = alloc_executable_memory(SIZE);
    LOGD("MALLOC␣ADDR:␣%p", m);
    emit_code_into_memory((unsigned char*)m);

    JittedFunc func = (JittedFunc) m;
    LOGD("FUNC␣ADDR:␣%p", &func);

    int a = 20;
    int b = 4;
    LOGD("Result␣of␣%d␣*␣%d␣=␣%d", a, b, func(a, b));
}
```

Note that the function typedef variable `func` is just treated as a variable, meaning its address lies on the stack (like it is supposed to be) and the `m`'s address allocated by `mmap()` is displayed in the `/proc/self/maps` output without any "`[libc_malloc]`" annotation or path but just left blank.

If the original C function does not contain any external library calls, the execution of machine code directly out of memory is pretty straight forward, at least for one specific function like demonstrated above for a multiplication. When library calls are performed inside of that method, it gets much more complicated. Even for function calls within the same C file it is not a simple task since the linker functionality is completely missing. Therefore, addresses to jump to, have to be adapted by hand.

So in general, executing code out of memory is possible in Android ART but to be able to use it productively, some more work has to be done in order to invoke whole shared objects or whole binaries instead of machine code chunks. Just implementing an ELF parser at runtime to extract all kinds of defined methods, would not be enough but additional alterations to the machine code need to be done to adjust jump addresses to point to the the manually allocated memory address containing the machine code.

4.5 Performance Comparison

Table 4.4 shows a performance comparison of described methods with regards to dynamic code loading via DEX, shared objects and executables using Java and C/C++. While DEX and the shared object rely on the same testing multiplication function, the tested executable implemented a `printf()`. That means, that the measured values do not make any statement about invoking executables versus invoking shared objects and DEX files but instead about the different

implementation. As expected, loading a method out of a DEX at runtime is significantly slower

Table 4.4: Dynamic Code Performance Comparison

Object to Load	Function	Loading Implementation	Time (ms)
DEX	mul(int, int)	Java	6-9
Shared object	mul(int, int)	Java	3-4
Shared object	mul(int, int)	C/C++	2-4
Executable	printf()	Java	23-27
Executable	printf()	C/C++	24-26

(takes nearly twice as long) than invoking from a shared object. The reason behind that is the compiling step that has to be performed in order to produce the native code ODEX since ART can not execute DEX byte instructions. However, the language used for invocation does not matter at all. That makes sense since Apps are being compiled by the ART VM at installation time and the output is native code in both cases (Java or C++). Differences should therefore only occur due to the implementation of the transformation step - meaning the compiler.

4.6 General Memory Access

In general, an App can write into every memory section which is mapped with writing permission. To find writable locations in the mapped space, the output of `/proc/self/maps` can be used via triggering the permission triple "`rw-p`" string. Listing 4.12 for instance triggers the specific writable `/system/bin/linker` entry. The size of the memory region can be parsed from the matching line. "`strtoll`" can be used to do the actual parsing from a hexadecimal char sequence to its "`long long int`" number to be casted to a pointer (=memory address) at the next step. Usually a char pointer is used for the actual reading/writing process since char always has the size of one byte. This way, when using array notation, each byte of the memory can be easily accessed.

Listing 4.12: memoryWriting()

```
JNIEXPORT void JNICALL Java_schleemilch_ma_nativememory_MyNDK_memoryWriting
 (JNIEnv *env, jobject obj){
    FILE * fp;
    char line[2048];
    fp = fopen("/proc/self/maps", "r");
    if (fp == NULL){
        LOGE("Could not open /proc/self/maps");
        return;
    }
    while (fgets(line, 2048, fp) != NULL) {
        if(strstr(line, "rw-p 0001d000 b3:19 366") != NULL){
            LOGD("%s", line);
            break;
        }
    }
    fp->_close;

    char address[9];
    strncpy(address,line,8);
    address[8] = '\0';

    long long int mp = (long long int)strtoll(address, NULL, 16);
    void* vp = (void*)mp;
    char* cp = (char*) vp;
    LOGD("%p", cp);

    LOGD("Print/Write Memory:");
    for (int i = 0; i < 10; i++){
        LOGD("%x", cp[i]);
        cp[i] = i; //writing
    }
    LOGD("Print Memory:");
    for (int i = 0; i < 10; i++){
        LOGD("%x", cp[i]);
    }

}
```

Being able to write to memory is generally a necessary precondition for advanced patching techniques like adapting the linker and is therefore doable in ART.

4.7 Utilizations

The question is how those gathered methods of dynamically loading code can be used practically. A few examples of how those dynamic code loading methods can be used are shown in the following. These methods are meant to inspire the reader to investigate even more advanced solutions.

4.7.1 Licensing Improving

Licensing is a common way of controlling additional paid features of an App. The main attacking point of licensing mechanisms is the one function call that checks the licensing status of its user. As pointed out in [Ber15], the call which checks the license can be easily patched. In many cases it is enough to use Lucky Patcher or similar software so that the function call automatically always returns "`licensed == true`" or gets skipped entirely. How to solve this problem? A good practice would be to deliver necessary parts of an App only when the licensing mechanism was called successfully, meaning that a connection was successfully established between the App and the licensing server that in turn will distribute the additional App content (see Figure 4.1). So even if

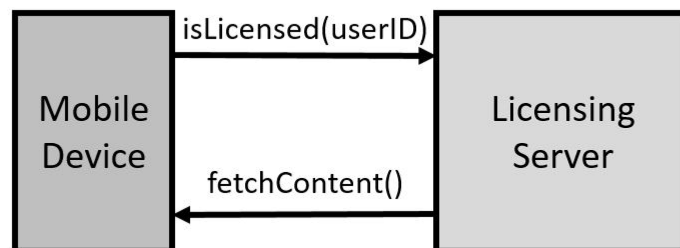


Figure 4.1: Dynamic App Content Loading

an attacker manages to skip the license call and trick the App into thinking it is licensed, he still would not have access to additional functionalities because of missing application files that will not be transmitted from the licensing server. As shown in this chapter, that kind of behavior could be implemented in native code as well as in Java and is also relatively independent in terms of the type of code to be loaded (DEX, shared library, binary). Nevertheless, if those additional App code pieces are being downloaded successfully, with root rights, they can be pulled out of the device. This means that an advanced attacker could again circumvent the licensing mechanism for several devices after purchasing the App once and afterwards patching the license/code fetching calls. A possible remedy for that issue could be continuous license checking calls that run in the background and which are implemented as a service whenever there is an Internet connection present (e.g. every 10 minutes). That service could not only delete files that should not be present on a certain device but could also lead to an App crash with a meaningless message in case of a missing legitimate license.

A simple way of crashing an application is to trigger a segmentation fault (SIGSEV) for instance by writing to the NULL-pointer. In C/C++ that is simple since pointers do exist (see Listing 4.13).

Listing 4.13: crashApp()

```
unsigned char *p = 0x00000000;
*p = 1;
```

That will cause an Android GUI message “<AppName> has stopped working”, but the ADB interface will reveal the actual cause “Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0 in tid 25271 (ma.nativememory)”. Additionally, debug information of all the CPU registers and their

Table 4.5: ADB SIGSEV Debugging

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'google/.../2554798:user/release-keys'
Revision: '0'
ABI: 'arm'
pid: 25515, tid: 25515, name:ma.nativememory
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
r0 b4d56a80 r1 bea34ffc r2 00430000 r3 00000000
r4 7045ea08 r5 12c5b800 r6 00000000 r7 00000000
r8 12d8e100 r9 b4d76500 sl 12c5b800 fp 6fe4d244
ip b3b01259 sp bea34ff0 lr a211dee1 pc b3b0125a cpsr 600e0030
#00 pc 0000125a /data/app/<AppName>/lib/arm/libMemory.so
(Java_<package>_MyNDK_crashApp+1)
#01 pc 0044eedf /data/app/<AppName>/oat/arm/base.odex
(offset 0x2d5000) (void <package>.MyNDK.crashApp()+74)
#02 pc 0059dc9b /data/app/<AppName>/oat/arm/base.odex
(offset 0x2d5000) (void <package>.MainActivity.onCreate(android.os.Bundle)+854)
#03 pc 72da30a9 /data/dalvik-cache/arm/system@framework@boot.oat
(offset 0x1ec9000)
```

contents as well as the program counters (pc) and their corresponding methods that lead to the crash will be displayed (output is shown in Table 4.5). With this information, an attacker can find the entry-point (=the method) that leads to the crash from a Java perspective (`onCreate() -> MyNDK.crashApp()`) as well as from a native perspective (library “libMemory.so” and function name “Java_..._crashApp()”). So in turn, he could again try to patch those function calls in order to prevent the App from crashing. Therefore, it is kind of a race between who can come up with the smarter solution, the attacker or the protector of an App.

4.7.2 String Encryption

Another idea of an dynamic code application would be the encryption of arbitrary strings that are used inside of an App and which makes the content unreadable.

First it will be investigated if those strings can be decrypted on the fly using native code after fetching the secret key from a trusted source (e.g. via HTTPS, a SE or the like). A precondition of that concept is being able to locate those defined strings in the process memory and access them using the NDK and therefore C/C++. The main idea is to define a priorly known value ("egg value") which is placed right before the actual string to locate it like shown in Listing 4.14 at Java layer because in the end a string are just bytes and can not be identified by itself.

Listing 4.14: Egg Value Defining

```
int egg = 0x11223344;
String anyString = "toBeEncrypted";
```

To find the egg value, all the mapped areas can be swept to find about their current addresses and triggering them to investigate the egg value as well as the actual string. The implementation is shown in Listing 4.15 where the mapped frame has to be parsed to find the memory range to sweep over.

Listing 4.15: Egg Value and String Memory Sweep

```
char adress[9];
FILE* fp;
char line[2048];

fp = fopen("/proc/self/maps", "r");
if (fp == NULL){
        LOGE("Could␣not␣open␣/proc/self/maps");
        return;
    }
long long int mp;
void* vp;
char* lowerLimit;
char* upperLimit;
while (fgets(line, 2048, fp) != NULL) {
    if(strstr(line, "rw-p") != NULL){
        strncpy(adress,line,8);
        adress[8] = '\0';
        mp = (long long int)strtoll(adress, NULL, 16);
        vp = (void*)mp;
        lowerLimit = (char*) vp;

        strncpy(adress,line+9,8);
        adress[8] = '\0';
        mp = (long long int)strtoll(adress, NULL, 16);
        vp = (void*)mp;
        upperLimit = (char*) vp;

        for (char* i = lowerLimit; i < upperLimit - 4; i++){
            if (i[0] == 0x44 && i[1] == 0x33 && i[2] == 0x22 && i[3] == 0x11){
                LOGD("FOUND␣EGG␣at␣%p",i);
```

```
                LOGD("%s",line);
            }
        }
        for (char* i = lowerLimit; i < upperLimit - 13; i++){
            if (i[0] == 't' && i[1] == 'o' && i[2] == 'B' && i[3] == 'e' && i[4]
                 == 'E' && i[5] =='n' && i[6] =='c' && i[7] =='r' &&
                i[8] =='y' && i[9] =='p' && i[10] =='t' && i[11] =='e' &&
                i[12] =='d'){
                    LOGD("FOUND STRING  at %p",i);
                    LOGD("%s",line););
                    //Change something:
                    i[0] = 'S';
            }
        }
    }
}
fp->_close;
```

This makes sense only to trigger for regions that are marked as "`rw-p`". Since values are stored as Little Endian, the egg value bytes have to be reverted. The string itself cannot be found this way (considered Little and Big Endian UTF-8, UTF-16 and Unicode). A reason could be that strings on Java layer as well as on C/C++ are no primitive data types and there is also the conversion step of `dex2oat`. However, that can be circumvented by using a byte array to store the string ("`byte[] strBytes = "toBeEncrypted".getBytes();`"). When doing so and running the program, the output will look like shown in Table 4.6.

Table 4.6: Egg String Hunting Output

```
FOUND EGG at 0x12c41390
12c00000-12de3000 rw-p 00000000 00:04 7019 /dev/ashmem/dalvik-main space
FOUND STRING at 0x12d3b22c
12c00000-12de3000 rw-p 00000000 00:04 7019 /dev/ashmem/dalvik-main space
```

It makes clear that the address difference between the egg value and the string is huge. Additionally, the difference between those addresses is not stable when rerunning the program so that the egg value misses its purpose completely (most likely due to Address Space Layout Randomization (ASLR) that is completely implemented in Android since version 4.1). Since a byte array is used, the egg value can be integrated into the array by simply merging them. So how to apply changes of the found string bytes to the Android App? Changing in C++ shouldn't challenge since array indexing can simply be used (e.g `i[0] = 'S'` when the string is found). Back in Java there are two options to apply the changes of used strings in `TextViews` and the like. The first option is calling the `setText()` function manually of every changed item and the second option is restarting the whole activity after the change occurred via `finish()` and

`startActivity(getIntent())`. The second option would need variables that are initialized as `static` to persist after the activity recreation. Otherwise the changes that have been made would be overwritten. Listing 4.16 shows a Java App implementation of a string that will be changed through native code as shown in Listing 4.15 (=function `eggHunting()`) and updated afterwards.

Listing 4.16: Native Code String Change

```
public class MainActivity extends AppCompatActivity {
    byte[] cbytes = "toBeEncrypted".getBytes();
    //static byte[] cbytes = "toBeEncrypted".getBytes();

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final MyNDK ndk = new MyNDK();
        final TextView eggText = (TextView)findViewById(R.id.textView);
        try{
            eggText.setText(new String(cbytes,"UTF-8"));
        } catch (UnsupportedEncodingException e){
            e.getMessage();
        }
        Button eggButton = (Button) findViewById(R.id.btn_change_egg);
        eggButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                ndk.eggHunting();
                try{
                    eggText.setText(new String(cbytes,"UTF-8"));
                } catch (UnsupportedEncodingException e){
                    e.getMessage();
                }
                //Alternatively...
                //finish();
                //startActivity(getIntent());
            }
        });
    }
}
```

The groundwork for an implementation is set. So the next thing to deal with is the encryption itself that can be implemented in very different ways. Strings defined on the Java layer could be encrypted in the Java world using for instance the `Cipher` module and AES. After the encryption, the egg bytes need to be added so that the native code can find those encrypted bytes and decrypt them on the fly. That concept however is a great detour and leads to issues like different AES implementations in Java/C++ as well as changing byte array sizes at Java layer. This is due to AES producing encrypted data that is always a multiple of the AES block size but the payload would most likely be smaller or greater. This means the Java array sizes won't match in the NDK world after the decryption and therefore the concept would be very fault-prone. The NDK could know the final array size from an additional value after the egg bytes, but the array size in Java would still be the one including the encrypted bytes. The principle is being shown in Figure 4.2.

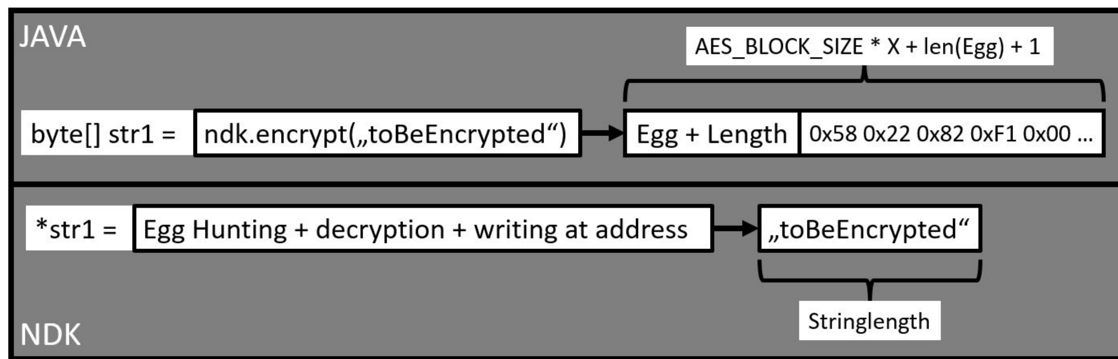


Figure 4.2: Dynamic Content Decryption using Eggs

Another approach would be to perform the whole encryption/decryption in native code while programming the logic in Java. The work flow for the developer would look like this: Strings are defined as byte arrays which are initialized by calling the corresponding NDK method for encryption and returning the encrypted bytes. App objects of the App can use those bytes as if they were strings by using a "`new String(bytes)`" call. After checking the license of the App, those byte arrays can be replaced with bytes returned by the NDK decryption method. Listing 4.17 shows the example App and the corresponding NDK calls to demonstrate their signature.

Listing 4.17: NDK Encryption/Decryption

```
static final MyNDK ndk = new MyNDK();
public class MainActivity extends AppCompatActivity {
    byte[] cbytes = ndk.encrypt("toBeEncrypted","secretkeyxxxxxxx");

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        final TextView eggText = (TextView)findViewById(R.id.textView);
        eggText.setText(new String(cbytes));

        //license check
        if (isLicensed()){
            eggText.setText(ndk.decrypt(cbytes,"secretkeyxxxxxxx"));
        }
    }
}
```

Listing 4.17 makes clear that one call (`isLicensed()`) is responsible for transforming an App to have readable content since the key is hard-coded in this case and would therefore be the weak spot in this approach. AES was used for encryption/decryption which is a symmetric method for a proof of concept. To make this technique more secure, one could fetch the decryption key inside of the `isLicensed()` method so that even when avoiding the license call, the text would still remain encrypted. This can be achieved by using an asymmetric encryption technique for

example "ElGamal". The encryption key could then be hard-coded since the private counterpart to decrypt would not be available.

Another solution would be to not encrypt those strings by calling a function but with another program/script and only assigning the outcome to byte arrays - the encrypted bytes. Then also a symmetric technique like AES can be used after fetching the decryption key when being licensed. In the authors opinion that would be the most secure solution since the string to be encrypted would not appear anywhere in the DEX or in memory as clear text (sweeping over the memory reveals that the string to be encrypted in the assignment `ndk.encrypt("toBeEncrypted","secretkeyxxxxxxx");`') which is present in memory and could also be present in the DEX). The downside of this approach is a bit of extra work for the developer since he would need to call an encryption script for every string he wants to use and has to copy those bytes into his Java source code. The NDK C++ AES encryption/decryption methods can be found in Appendix B.

OpenSSL libraries are present in Android but are not intended to be used by the NDK. So it can either be built manually as explained in [Wik16] or a precompiled version can be used like the one compiled in [emi16]. Android Studio's makefiles needs to be adapted in order to link those libraries. The AES implementations shown in Appendix B are based on the precompiled version (it's precompiled for armeabi-v7a as well as for x86).

Overall, string encryption can be implemented quite secure. The question is if strings are really a crucial part of an App that needs to be protected. Generally, this technique is not limited to strings but can also be applied to any other App content. Implementing crucial security mechanisms in native code like for example fetching decryption keys or doing the actual encryption/decryption should increase the reverse code engineering resistance. Due to time constraints an in-depth evaluation cannot be performed. Of course the NDK is not mandatory for such a encryption/decryption technique but adds another layer of complexity for an attacker to cope with.

4.8 Decompilation

From an attacker's point of view it is interesting to know if and how those dynamic code loading techniques can be decompiled. One of the best decompilers for ELF files and therefore theoretically also for ART ODEXs as well as for shared libraries is the "Retargetable Decompiler" available at [Kro16]. The decompiler is able to decompile machine code into assembly as well as into C and Python source code. Additionally, control flow graphs for every method can be viewed and

downloaded. The author offers a web interface for uploading and decompiling files. However, the tool crashes when trying to decompile whole ART ODEX Apps. Having said that, it would also not be very useful since DEX decompiling is much more reliable to produce Java code that enables the understanding of the main App structure (assuming it is not a purely native App). For native parts though, DEX decompiling will not be sufficient. First, those machine code files (libraries, executables) need to be obtained from the device. If the NDK is used and files to be reviewed are statically embedded, they can be pulled via the ADB and device root rights by first copying those files to the `/sdcard/` path and finally using `adb pull`.

The decompiled code is good in general but naturally more time consuming to understand than its source code counterpart. Let's take a look at the "`encrypt()`" function introduced in subsection 4.7.2 and delineated in Appendix B. It has been chosen as reference function since it uses the JNI as well the external AES library OpenSSL. Listing 4.18 shows the two signatures of the function in original and decompiled source code.

Listing 4.18: Original vs Decompiled JNI Method Signatures

```
//Source...
JNIEXPORT jbyteArray JNICALL Java_schleemilch_ma_nativememory_MyNDK_encrypt (
    JNIEnv *env, jobject obj, jstring str, jstring jkey);
//Decompiled...
int32_t Java_schleemilch_ma_nativememory_MyNDK_encrypt(struct struct_6 a1,
    int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6, int32_t a7,
    int32_t a8, int32_t a9, int32_t a10, int32_t a11);
```

The first thing that attracts attention is that method names are maintained completely which offers semantic information. A different story is the signature itself. While the original function returns a byte array, the decompiled version only returns a 32-Bit integer value. The same thing holds for the transferring parameters which cannot be recognized as strings. Structs handed over like "`struct_6`" are defined on top of the file but then again only contain several `int32_t` values. CPU registers and flags are represented by global variables starting with a "g" and are associated with their names (R0, R1, LR, SP, . . .). Since function names are equal to their original counterparts, calls using "`env`" like "`const char *input = env->GetStringUTFChars(str, NULL);`" including its parameters, can be interpreted quite good. The same goes for library calls like "`AES_set_encrpyt_key()`" but in this case without parameters. The full decompilation of this native encrypt function can be viewed in Appendix B.

So in general it is possible to understand those decompiled versions of source code especially if a lot of function calls were used which helps to interpret the overall semantics. Especially control flow graphs can help understanding the fundamental code intention. For untrained readers though, the decompiled version looks very messy juggling with registers. A well versed

programmer (maybe with assembler background), should be able to understand the code much faster.

Going back to the develops that aim at protecting their source code, the author would suggest to prevent attackers from obtaining crucial files in the first place, either by using encryption or dynamically loading files. This approach is especially reliable if preconditions are fulfilled, but not necessarily if an App needs to work reliable without internet access.

Chapter 5

ART Native Code Store

Since DEX code seems to be the vulnerable spot for Apps in terms of security topics as well as for licensing and piracy issues, it makes sense to try to circumvent this file format by design. When comparing the mobile device distribution system of Apps to desktop environments like Linux, Windows or MacOS, it becomes clear that the main difference is the distribution of Java like byte-code compared to binaries, at least for commercial software or the operating system itself. So the question would be if it is possible to establish an App store that only distributes native code instead of APKs. As derived in subsection 3.1.2, it is more than very likely that the DEX file is still needed for addressing native code inside of an ELF. Nevertheless, that thought experiment will be performed in order to investigate if it would generally be possible.

At first, an architecture that would be suitable for an alternative native code App store needs to be defined. To keep the user experience as is, it would be beneficial to be able to still use the Google Play App Store for distributing Apps. Therefore at least a bare-bone APK for the application needs to be created that will then be placed into the Google Store. That application can be installed the common Android way. At its first startup, it could connect to the actual native code App store requesting the functional App in form of raw byte code or an ELF file that somehow gets injected into the current bare-bone application. Since the Android ODEX ELF file of an App has the DEX embedded, it would need to be adapted for instance leaving the DEX area blank. So the skeleton version needs to implement at least the communication mechanism as well as the self modifying code part that can handle the receiving code snippets (Figure 5.1 visually shows the explained architecture). When assuming that no root rights are present, the possibilities of injecting code and changing files are of course very limited. Remember that there exist two identical DEX files after the installation process (post Android 5), one DEX inside of the original `base.apk` package and one embedded inside the `dex2oat` output (ELF file). Let's see to whom those files belong at Linux layer and what permissions are set. Files stored at `/data/app` like the

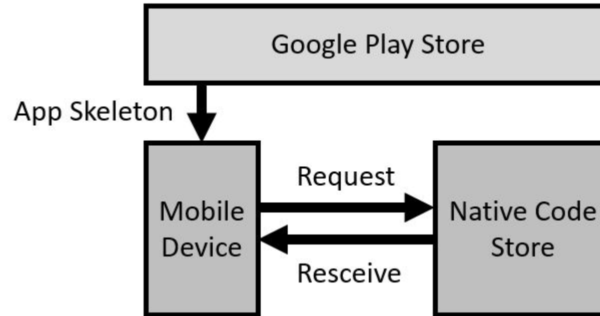


Figure 5.1: Native Code Store Architecture

base.apk, the lib/ folder as well as oat/ belong to the system user. So accessing the base.apk container including its DEX should be impossible from the App context without root rights. The same holds for the base.odex which also belongs to system and is marked as "rw-".

Another way to check the possibilities of changing files dynamically at runtime are the mapped files and their permissions that can be read out of /proc/self/maps using C/C++ as explained in subsection 4.4.1. Remember that "mapped" means that those files are copied into the memory and therefore changes made in mapped files are not taken over by the actual physical stored counterparts. So changes written to mapped files would not be permanent. Since the last entity of an App getting executed is the ODEX file, it is mapped into the process (Table 5.1 shows an example). Several regions of the ODEX are mapped (offset marks the beginning relative to its file)

Table 5.1: App's ODEX/ELF mapping

address	perms	offset	dev	inode	rel. pathname
a1d05000-a1fda000	r-p	00000000	b3:1c	171546	app/.../oat/arm/base.odex
a1fda000-a22ab000	r-xp	002d5000	b3:1c	171546	app/.../oat/arm/base.odex
a22ab000-a22ac000	rw-p	005a6000	b3:1c	171546	app/.../oat/arm/base.odex
a22ac000-a2327000	r-p	006ed000	b3:1c	171546	app/.../oat/arm/base.odex

with different permissions. Interesting is the part including writing permissions and its content. The corresponding ELF format was analyzed in section 3.1.1. To find out which specific section is mapped as writable, a program can be written in order to dump the content of every section including its offset. Other options would be to just use a hex editor or even the readelf tool. So first, the ODEX has to be pulled out of the Android device for which root rights are required (otherwise it is not possible to even navigate into the App directory). The file can then be moved into the /sdcard path and copied from a computer using adb pull /sdcard/base.odex. It then

becomes clear that the mapped file region marked as writable is located in the string table section "`.strtab`". That mapped section though is not useful in terms of altering the Apps own code since it does not contain the DEX nor the ELF file. Therefore, a concept of distributing native code in form of cross-compiled ELF files without root rights is not possible.

Chapter 6

Trusted Execution Environments (TEEs)

Trusted Execution Environments do have a great potential for achieving a new security level for mobile devices. In [Jan13], an introduction to TEEs for the mobile domain is given and will be recapped at this point. TEEs offer a parallel execution environment for applications including storage. A crucial part is the isolation of the TEE from its counterpart, often called the Rich Execution Environment (REE) running on top of an underlying Rich OS (Android, Windows OS, Symbian OS). The main goal is to outsource sensitive operations from the REE to the TEE so that sensitive data won't leave the secure world and to prevent malicious code from the Rich OS to interfere with this secure environment. Security applications often make usage of tokens (hardware and software based, one-time tokens and two-factor authentication). TEEs do have the potential of replacing them also with an increase of usability for users. Today, even every smartphone includes a TEE. Since ARM CPUs are quite popular in mobile devices, it is likely to find an ARM TrustZone TEE but there also exist implementations for Intel (TXT) and AMD (Secure Execution Environment) that do behave quite similar. A general TEE overview is shown in Figure 6.1. TEEs can therefore be seen as a direct copy of the common system, meaning that it has for instance also its own Thread and Handler modes.

There do exist two main TrustZone technology concepts - the ARM TrustZone Technology for Cortex-A and Cortex-M Processors as well as the ARM TrustZone CryptoCell [ARM16b]. As mentioned before, the main concept of TEEs is to separate two worlds, for instance by choosing an own physical CPU for each of them. A switch between them can be applied using a secure monitor (application processor) or the hardware (microcontrollers). The separation however must not stop with CPU separation but can also be applied to memory and software itself. The CryptoCell expands the security possibilities by providing hardware support for the acceleration of security processes. It includes more efficient cryptographic engines, secure boot options as well as a root

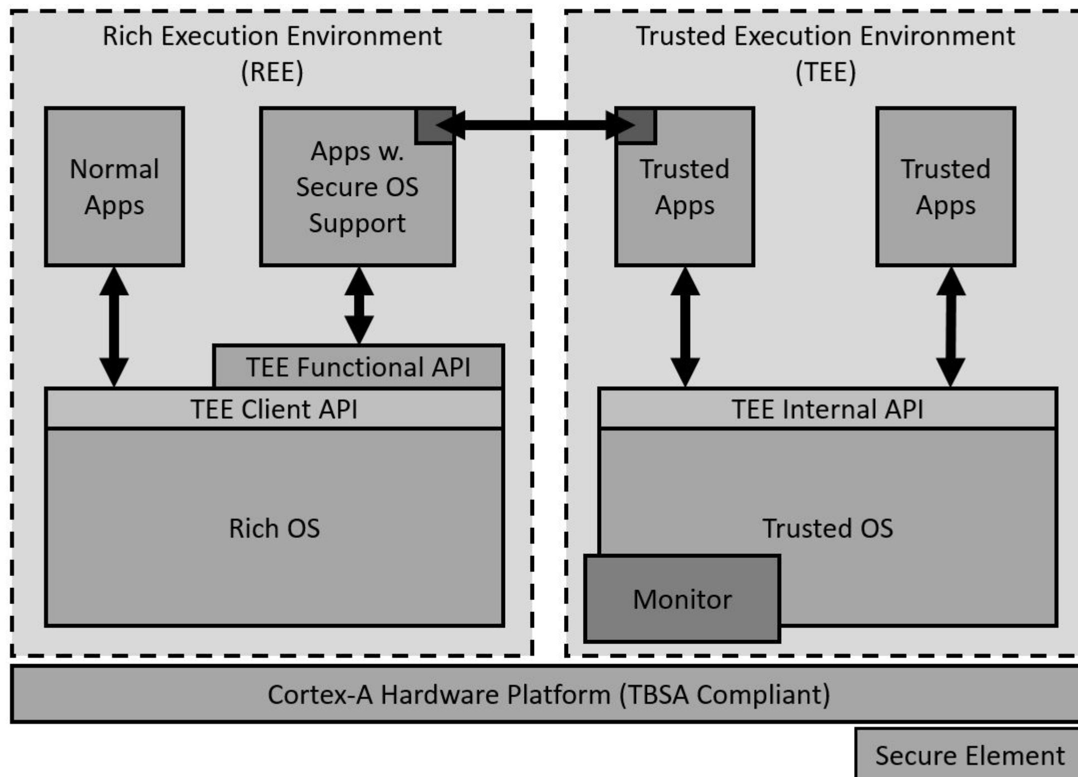


Figure 6.1: General TEE Structure taken from [ARM16a]

of trust element including key management. It acts as a completely separate functional block to a CPU. Let's summarize the buzzwords like proposed in [Glo16]:

- **Rich OS** is the common environment/operating system of a mobile device which focuses on applications and functionality but with a secondary concern for security matters. It is therefore open for third party applications that can be downloaded by the user.
- **TEE** is built on top of the Rich OS and offers a trusted environment using software but most importantly also hardware to achieve that goal. Only trusted applications are allowed to execute code inside of that environment. Is is designed to overcome software attacks which try to spread over from the Rich OS.
- **Secure Element (SE)** is a piece of hardware that is tamper-resistant within secure applications in which secure data can be stored. It has very limited functionality while offering a high level of security.

Some relations between those technologies and their trade offs are shown in Table 6.1. The Rich OS is trimmed for usability and therefore it has the best user interface support, it is easy to develop Apps and the processing speed is as good as it can be on the specific hardware. What

Table 6.1: Rich OS, TEE and SE Comparison taken from [Glo16]

Performance/Comfort			
Technology	User Interface	Ease of Development	Processing Speed
Rich OS	++	++	++
TEE	+	o	++
SE	-	o	o
Security/Flexibility			
Technology	Attack Resistance	Access Control	Phys. Removable
Rich OS	–	-	n.a
TEE	+	+	n.a
SE	++	++	++

the TEE is lacking compared to the Rich OS is the ease of developing Apps and part of the user interface possibilities. However, the processing speed is quite the same since there is no additional abstraction layer but only a hardware separation. The Secure Element cannot be controlled via an UI, it is not very intuitive to develop applications and the processing speed is limited to the SE hardware which has likely the worst performance. When looking from a security perspective, the Rich OS is as expected not as good as the comparing technologies that are designed especially for that purpose. So the attack resistance is low compared to TEEs and SEs and is limited to SE Linux and Android functionalities as well as to the Access Control. Physical removability does not make sense for an operating system as well as for TEEs but is a benefit of SEs which are the most secure solution.

TEEs are a great concept offering an environment that can be trusted for sure. For common developers however it is not that simple to make use of TEEs. The crucial part is to be able to write trusted applications (TAs) that are allowed to be run in the TEE. Right now, a fee must be paid to receive the necessary framework for writing those Apps. So in theory it is a nice concept but practically it lacks in terms of usability for common developers due to the fee barrier. TEEs are based on ensuring that no malicious developer/App resides in its environment. However if TEEs will open up to more and more developers, that will again be hard to guarantee.

Chapter 7

Related Work

[Jeo12] introduces an anti-piracy mechanism that is based on class separation and dynamic loading at Java level. The main concept is to separate the app source code into an "Incomplete Main Application" (IMA) and a "Seperate Essential Class" (SEC) that gets downloaded at first use of the app and is only decrypted after authentication. For loading, common `DexClassLoading` is used as shown in section 3.3.2. This method is still possible after the ART transition but shas the downside of enabling reverse code engineering of the additional SEC element after a one time App execution. It should be vulnerable versus dynamic reverse code engineering.

[Fal15] addresses the problem of developers implementing unsafe variants of dynamic code loading techniques by calling common Android APIs like `dexClassLoader`. The result is a wrapper (`SecureDexClassLoader`) for dynamic code loading techniques again at Java layer that includes security checks for fetching code from an URL, storing the code in an app-private directory, ensuring integrity and developer authenticity of the code, and finally load it. It is a useful practical tool for developers who are not familiar with all kinds of security risks associated with using dynamic code loading techniques. In the scope of copy protection mechanisms, it is not that revolutionary though but could be used when implementing a dynamic code loading mechanism on Java layer.

Since dynamic DEX loading is still possible, Android Packers might still be a good choice of protecting intellectual property. In [Yu14] an introduction to Android Packers is given, explaining the difference between packers and obfuscation as well as introducing popular packers like ApkProtect, Bangcle and Iliami and addressing future challenges in the packer domain. [Yue14] on the other side analyzes the major techniques used by the most common packers also addressing ART as well as Dalvik. The novel system "DexHunter" they developed is able to recover most DEX files so their work indicates that packaging services are not as secure as they appear to be.

Talking about the protection of intellectual property, [San12] shows a concept of inserting a forensic mark to an App that inserts buyers information right into the `classes.dex`. It then describes a technique of verifying the app license with the mark as a foundation. This technique could be added to a license mechanism to increase its robustness.

[Ash15] proposes a visual method of analyzing Android executable files including ART to reveal patterns. Although it focusses mainly on finding anomalies in malicious Apps, it could also be used to classify common Apps and possibly gain additional information with less effort compared to other reverse code engineering techniques. The authors are parsing the DEX file and coloring its file structure, transfering the gained information to its corresponding binary.

[Tim14] explores secure key storage options in Android that are needed for instance for encryption/decryption scenarios. It compares the built in feature to the Bouncy Castle key storage solution. The security of the built in feature depends on the device and might not make use of ARM TrustZone features. Bouncy Castle on the other hand can provide even stronger security guarantees.

Since the Google Play Store aims to be malware free using the Google Bouncer that scans new Apps for malicious code, [Dom] shows that it can be surpassed using a technique called "Divide-and-Conquer" that is basically built on using dynamic code loading. The authors are concluding that neither static nor dynamic analysis is sufficient since attackers can write malware that behaves differently in analysis environments. They are called "split-personality" malware. Anti-virus scanners can not detect them at runtime since monitoring of third party Apps at runtime is not possible.

Already in 2008 there was a concept quite similar to todays TEEs that is described in the paper [Jon08] and is called "Flicker". A system is introduced that offers complete isolation from the Rich OS (also from a hardware perspective) while trusting a very small code base. It can be seen like a predecessor to TEEs.

"VirtualSwindle" is an App described in [Col14] that aims at automatically attacking in-app purchasing with the goal of accessing the content the user should purchase for free. A specific implementation of purchasing mechanism is being attacked in a Dalvik environment. Instead of revers engineering Apps to be patched, a system is developed that can inject arbitrary code into a process. The introduced app runs in the background and attacks every App using the in-app billing mechanism. Authors do make clear that their App does not reveal a weakness in Google's in-app billing mechanism but in lazy implementations of developers. However, since it has been developed for Dalvik byte-code, it should not be possible to use it under ART anymore.

In [Seb16] it is analyzed how effective obfuscation techniques are in comparison to their counterpart, the code analysis and if it can keep up with its developing pace. It makes clear that it is still an arms race between software developers and code analysts. A result indicates that the effectiveness of obfuscation highly depends on the analyst and his available resources (computational and financial). So the arms race between those two groups persists and obfuscation versus a human analyst is an ongoing challenge. If an attacker has enough time and resources he will very likely be able to reveal the actual program flow and intention.

Root checks are a possibility to add an additional layer of security by preventing App installations when a device is rooted. Despite the fact that it may offend power users, [Nat15] shows possible circumventions for those root checks. The tool "AndroPoser" is introduced which can suppress root checks and can make rooted devices appear as if they were non-rooted devices. Root checks often rely on path checks of tools that are only present when a device is rooted (like the tool "su"). One very simple solution is to just rename that "su" binary. Other root check methods and their circumventions are being described.

Chapter 8

Conclusion

The runtime transition from the Android specialized virtual machine (Dalvik VM) to the Android Runtime ART, introduced as an option in Android 4.3, did not change the difficulty of reverse engineering Android Apps. So they are in general still just as vulnerable to patching, theft and code injection as they were before.

However, internal mechanisms of ART do change the feasibility of copy protection mechanisms which are based on byte code interpreted by the virtual machine, like hiding whole methods in the DEX or inserting Junk-Bytes.

Since Apps still are distributed via the DEX format, the file format is still very present under ART. What the runtime transition changed, is the optimization step, that now produces an ELF file with AOT compilation, compared to the optimized DEX (ODEX) that relied on JIT. Although ELFs can normally run by themselves or are getting linked in another executable, the invocation of methods still happens through DEX. Dynamic obfuscation techniques regarding DEX files are still applicable in general. But it has to be kept in mind, that there is a conversion step between loading that file and execution. That could lead to performance issues depending on the file size.

The possibility of using C/C++ via the Android NDK is a powerful tool to dynamically modify the own code or loading additional content. Dynamic shared object loading from a file, as well as executing an external binary, can be accomplished using Java or the NDK. For security reasons, it would be interesting to dynamically load code directly out of memory, instead of writing it to a file first. This is generally possible since memory can be allocated by the App as well as marking its content as executable. As shown, that technique can be used to execute machine code directly. Invoking whole file formats is more complex, since linker functionalities are missing and need to be implemented manually. One idea was to distribute only native code instead of the reverse engineering prune DEX format. This concept did not work due to missing permissions.

The NDK provides a great contribution to copy protection mechanisms. Licensing can be improved by concepts that make the license call mandatory for the App to work properly. License mechanisms could include key fetching to decrypt App content.

Trusted execution environments are a great idea that has disadvantages in terms of usability for everyday developers. However, the can be used to make safe license calls or to stream content. A paid fee is required to write trusted Apps.

So, to achieve a secure copy protection mechanism, individual solutions using dynamic code or encrypting content are promising but key fetching mechanisms as well as the key storage have to be chosen wisely.

Nevertheless, it has to be kept also in mind, that obfuscation techniques, especially those that are dynamically loading code, can be abused also by malware to hide its true intention from static analysis tools.

8.1 Future Work

To provide a secure copy protection mechanism, some more work has to be done and it is an ongoing task. It could be analyzed in greater detail than described in section 3.1, how the method invocation of ELF methods into the forked Zygote process works. This may reveal new possibilities of circumventing the DEX format or of sparing out crucial parts that are not needed.

Possibilities of dynamic native code shown here are proof of concepts and would need to be used adequately. Especially more clever techniques regarding the licensing mechanism using cross-interlocking of native and Java code could be analyzed. Also asymmetric encryption as well as key handling needs more investigation.

A promising future work could include loading common files like executables right into memory and executing them. That would introduce a new layer of complexity from an attackers perspective, since there won't be any files residing on the file-system. Finally, the security (reverse engineering capability) of Java calls compared to Java calls through the JNI also needs further evaluation.

Appendices

Appendix A

NDK Project

Figure A.1 shows the interesting part of the Android Studio project tree with parts that has to be changed in order to make it work.

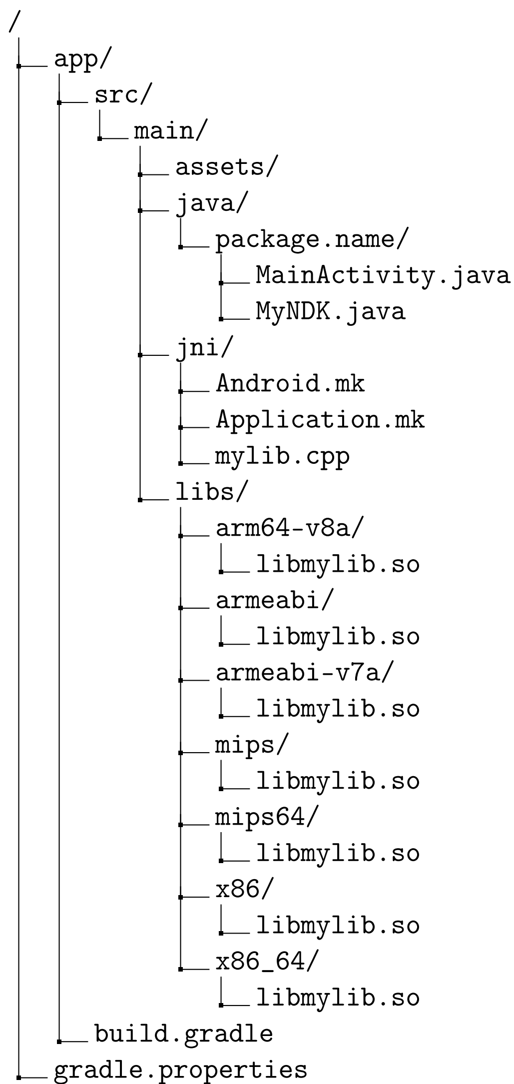


Figure A.1: NDK Project Tree Cutout

Listing A.1: MainActivity.java

```
package ma.schleemilch.nativestuff;

import android.content.Context;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    public static String TAG = "MYLOG";

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

        MyNDK ndk = new MyNDK();
        ndk.showNativeMessage("Success");
    }
}
```

Listing A.2: MyNDK.java

```
package ma.schleemilch.nativestuff;

public class MyNDK {
    static {
        System.loadLibrary("MyLib");
    }
    public native void showNativeMessage(String msg);
}
```

Listing A.3: mylib.cpp

```
#include "ma_schleemilch_nativestuff_MyNDK.h"
#include <string.h>

#include <android/log.h>

#define LOG_TAG "MYLOG"

#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

JNIEXPORT void JNICALL Java_ma_schleemilch_nativestuff_MyNDK_libExe
        (JNIEnv * env, jobject jobj, jstring msg){
                const char *msg = env->GetStringUTFChars(msg, NULL);
        LOGD("My native message: %s", msg);
}
```

Listing A.4: Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := mylib
LOCAL_SRC_FILES := mylib.cpp
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

Listing A.5: Application.mk

```
APP_MODULES := mylib
APP_ABI := all
```

Just showing the adapted part inside of `defaultConfig{...}`:

Listing A.6: build.gradle

```
ndk {
        moduleName "schleemilch"
}
sourceSets.main {
        jni.srcDirs = []
        jniLibs.srcDir "src/main/libs"
}
```

Appendix B

NDK AES Implementation

Listing B.1: AES Encrypt()

```
#include "openssl/aes.h"
uint8_t iv[16] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                   0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10};
uint8_t inputslength;

JNIEXPORT jbyteArray JNICALL Java_schleemilch_ma_nativememory_MyNDK_encrypt (
    JNIEnv *env, jobject obj, jstring str, jstring jkey){
    const char *input = env->GetStringUTFChars(str, NULL);
    const char *tkey = env->GetStringUTFChars(jkey, NULL);
    int keylength = env->GetStringLength(jkey)*8;
    uint8_t key[keylength/8];
    memcpy(key, tkey, keylength/8);

    uint8_t aes_key[keylength/8];
    memset(aes_key, 0, keylength/8);
    inputslength = env->GetStringLength(str);
    uint8_t aes_input[inputslength];
    memcpy(aes_input,input,inputslength);

    uint8_t iv_enc[AES_BLOCK_SIZE];
    memcpy(iv_enc,iv,AES_BLOCK_SIZE);

    const size_t encslength = ((inputslength + AES_BLOCK_SIZE) / AES_BLOCK_SIZE
        ) * AES_BLOCK_SIZE;
    unsigned char enc_out[encslength];
    memset(enc_out, 0, sizeof(enc_out));

    AES_KEY enc_key, dec_key;
    AES_set_encrypt_key(aes_key, keylength, &enc_key);
    AES_cbc_encrypt(aes_input, enc_out, inputslength, &enc_key, iv_enc,
        AES_ENCRYPT);

    jbyteArray ret = env->NewByteArray(AES_BLOCK_SIZE);
    env->SetByteArrayRegion(ret,0,AES_BLOCK_SIZE, reinterpret_cast<jbyte *>(
        enc_out));
    return ret;
}
```

Listing B.2: AES Decrypt()

```
JNIEXPORT jbyteArray JNICALL Java_schleemilch_ma_nativememory_MyNDK_decrypt (
    JNIEnv *env, jobject obj, jbyteArray jencrypted, jstring jkey){
    const char *tkey = env->GetStringUTFChars(jkey, NULL);
    int keylength = env->GetStringLength(jkey)*8;
    uint8_t key[keylength/8];
    memcpy(key, tkey, keylength/8);

    uint8_t aes_key[keylength/8];
    memset(aes_key, 0, keylength/8);

    uint8_t iv_dec[AES_BLOCK_SIZE];
    memcpy(iv_dec,iv,AES_BLOCK_SIZE);

    unsigned char dec_out[inputslength];
    memset(dec_out, 0, sizeof(dec_out));

    const size_t encslength = ((inputslength + AES_BLOCK_SIZE) / AES_BLOCK_SIZE
        ) * AES_BLOCK_SIZE;
    unsigned char enc_out[env->GetArrayLength(jencrypted)];
    env->GetByteArrayRegion(jencrypted,0, sizeof(enc_out), reinterpret_cast<
        jbyte*>(enc_out));

    AES_KEY dec_key;
    AES_set_decrypt_key(aes_key, keylength, &dec_key);
    AES_cbc_encrypt(enc_out, dec_out, encslength, &dec_key, iv_dec, AES_DECRYPT
        );

    jbyteArray ret = env->NewByteArray(sizeof(dec_out));
    env->SetByteArrayRegion(ret,0, sizeof(dec_out), reinterpret_cast<jbyte *>(
        dec_out));
    return ret;
}
```

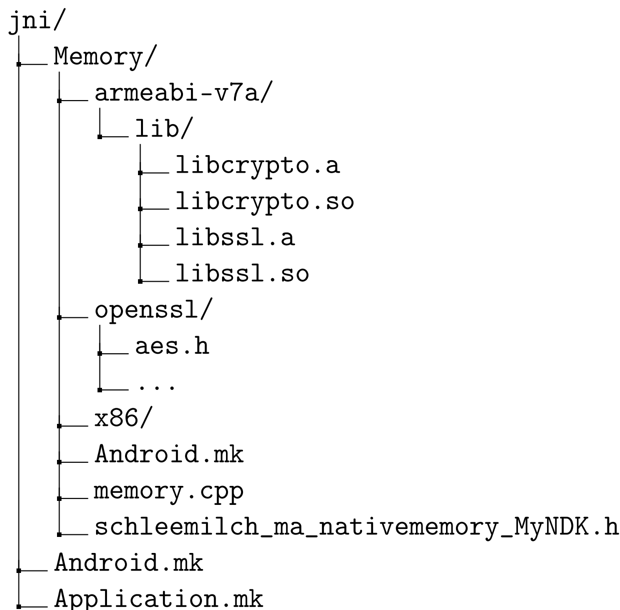


Figure B.1: NDK AES Implementation JNI Tree

Listing B.3: Memory/Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := opencrypto_static
LOCAL_SRC_FILES := $(TARGET_ARCH_ABI)/lib/libcrypto.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := Memory
LOCAL_SRC_FILES := memory.cpp
LOCAL_LDLIBS := -llog
LOCAL_C_INCLUDES:= openssl
LOCAL_SHARED_LIBRARIES := opencrypto_static
include $(BUILD_SHARED_LIBRARY)
```

Listing B.4: Decompiled AES Encrypt()

```
int32_t Java_schleemilch_ma_nativememory_MyNDK_encrypt(struct struct_6 a1,
    int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6, int32_t a7,
    int32_t a8, int32_t a9, int32_t a10, int32_t a11) {
    struct struct_6 v1; // 0x1560_7
    struct struct_6 v2; // 0x1594_8
    g32 = a4;
    struct struct_6 * v3;
    g34 = (int32_t)&v3;
    g35 = *(int32_t *)0x6f78;
    g33 = a3;
    g29 = 0;
    g2 = false;
    g4 = true;
    _ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh((struct struct_6 *)a1.e0, (char
         *)a3);
    g2 = false;
    g4 = true;
    int32_t v4 = g31; // 0x1552
    _ZN7_JNIEnv17GetStringUTFCharsEP8_jstringPh((struct struct_6 *)v4, (char *)
        g32);
    g26 = v4;
    int32_t v5 = g31; // 0x155e
    v1 = (struct struct_6){
        .e0 = 0,
        .e1 = 0
    };
    v1.e0 = v5;
    _ZN7_JNIEnv15GetStringLengthEP8_jstring(v1, g32, g5);
    g32 = v5;
    int32_t v6 = v5 + 7 & -8; // 0x156c
    g36 = v6;
    int32_t v7 = g37 - v6; // 0x1572
    memcpy((char *)(v7 + 8), (char *)g26, v5);
    int32_t v8 = v7 - g36; // 0x157c
    g37 = v8;
    int32_t v9 = v8 + 8; // 0x1580
    g26 = v9;
    int32_t v10 = g32; // 0x1584
    g2 = false;
    g4 = true;
    memset((char *)v9, g5, v10);
    int32_t v11 = g33; // 0x158e
    int32_t v12 = g31; // 0x1590
    g33 = 0x59e0;
```

```
v2 = (struct struct_6){
    .e0 = 0,
    .e1 = 0
};
v2.e0 = v12;
_ZN7_JNIEnv15GetStringLengthEP8_jstring(v2, v11, v10);
int32_t v13 = g33 + 0x159c; // 0x1598
g33 = v13;
int32_t v14 = *(int32_t *)v13; // 0x159a
g33 = v14;
int32_t v15 = v12 % 256; // R11
*(char *)v14 = (char)v12;
int32_t v16 = g37 - (v15 + 7 & 504); // 0x15ac
int32_t v17 = v16 + 8; // 0x15b2
g36 = v17;
memcpy((char *)v17, (char *)v3, v15);
int32_t v18;
int32_t v19 = &v18; // 0x15c0_0
int32_t v20 = *(int32_t *)0x6f80; // 0x15ca
g35 = v19;
int32_t v21 = v20; // 0x15d2
// branch -> 0x15d2
while (true) {
    int32_t v22 = v21 + 8; // 0x15d4
    *(int32_t *)v19 = *(int32_t *)v21;
    *(int32_t *)(v19 + 4) = *(int32_t *)(v21 + 4);
    int32_t v23 = v19 + 8; // 0x15de
    g28 = v23;
    g23 = v23;
    if (v22 == v20 + 16) {
        int32_t v24 = v15 + 16 & 496; // 0x15ec
        int32_t v25 = v16 - v24; // 0x15f0
        g37 = v25;
        int32_t v26 = v25 + 8; // 0x15f4
        g27 = v26;
        memset((char *)v26, g5, v24);
        g25 = 8 * g32;
        int32_t v27;
        int32_t v28 = &v27; // 0x1600_0
        g32 = v28;
        g24 = g26;
        g29 = v28;
        AES_set_encrypt_key();
        g29 = (int32_t)*(char *)g33;
        *(int32_t *)(g37 + 4) = 1;
        g30 = v28;
        *(int32_t *)g37 = g35;
        AES_cbc_encrypt(g36, g27);
        int32_t v29 = g31; // 0x1620
        g25 = 16;
        g2 = false;
        g4 = false;
        g24 = v29;
        int32_t v30 = *(int32_t *)(*(int32_t *)v29 + 704); // 0x1626
        g30 = v30;
        g23 = 0x162d;
        ((int32_t (*)())(v30 & -2))();
        int32_t v31 = g31; // 0x162c
        g29 = 0;
        int32_t v32 = *(int32_t *)(*(int32_t *)v31 + 832); // 0x1634
        g33 = v32;
        g30 = 16;
        g2 = false;
```

```
            g4 = false;
            int32_t v33 = g24; // 0x163a
            g32 = v33;
            g24 = v31;
            g25 = v33;
            g23 = 0x1643;
            ((int32_t (*)())(v32 & -2))();
            int32_t v34 = g34; // 0x1642
            int32_t v35 = *(int32_t *)(v34 + 4); // 0x1642
            g25 = v35;
            uint32_t v36 = *(int32_t *)(v34 + 268); // 0x1644
            g29 = v36;
            g24 = g32;
            int32_t v37 = *(int32_t *)v35; // 0x164a
            g30 = v37;
            uint32_t v38 = -2 - v37 + v36; // 0x164c
            g3 = ((v38 ^ v36) & (v38 ^ -v37)) < 0;
            g2 = v36 - v37 < 0;
            g4 = v36 == v37;
            g1 = v38 <= v36;
            int32_t v39; // 0x1654
            if (v36 != v37) {
                // 0x1650
                __stack_chk_fail();
                v39 = g34;
                // branch -> 0x1654
            } else {
                v39 = v34;
            }
            // 0x1654
            g31 = *(int32_t *)(v39 + 276);
            g32 = *(int32_t *)(v39 + 280);
            g33 = *(int32_t *)(v39 + 284);
            g34 = *(int32_t *)(v39 + 288);
            g35 = *(int32_t *)(v39 + 292);
            g36 = *(int32_t *)(v39 + 296);
            g26 = *(int32_t *)(v39 + 300);
            g27 = *(int32_t *)(v39 + 304);
            g37 = v39 + 312;
            ((int32_t (*)())*(int32_t *)(v39 + 308))();
            return *(int32_t *)(g31 + g34);
        }
        // 0x15d2
        v19 = v23;
        v21 = v22;
        // branch -> 0x15d2
    }
}
```

List of Figures

2.1 Android vs Linux . . . 5
2.2 APK Content Tree . . . 8
2.3 App installation process . . . 9
2.4 App execution process . . . 11

3.1 ELF file format . . . 13
3.2 OAT format . . . 16
3.3 DEX format . . . 17
3.4 ART App executable format . . . 18
3.5 Zygote Forking . . . 24
3.6 DEX Assembly/Disassembly . . . 26
3.7 Hidden Methods Invocation . . . 29

4.1 Dynamic App Content Loading . . . 49
4.2 Dynamic Content Decryption using Eggs . . . 54

5.1 Native Code Store Architecture . . . 59

6.1 General TEE Structure . . . 62

A.1 NDK Project Tree Cutout . . . ii

B.1 NDK AES Implementation JNI Tree . . . vi

List of Tables

3.1 ELF section headers . . . 14

3.2 ELF program headers . . . 15

3.3 ELF section segment mapping . . . 15

3.4 Android Processes . . . 19

3.5 Process Executables . . . 20

3.6 Arguments Class Attributes . . . 22

3.7 Zygote/App Start AOSP(6.0) Files . . . 25

4.1 Content of /proc/<PID>/maps . . . 41

4.2 Memory Allocation Mapping . . . 43

4.3 arm-none-eabi-objdump -D machineCodeMul.o . . . 44

4.4 Dynamic Code Performance Comparison . . . 47

4.5 ADB SIGSEV Debugging . . . 50

4.6 Egg String Hunting Output . . . 52

5.1 App's ODEX/ELF mapping . . . 59

6.1 Rich OS, TEE and SE Comparison . . . 63

Listings

3.1 ZygoteInit main loop . . . 21
3.2 Zygote Fork Call . . . 23
3.3 Zygote Child Loop Breakout Exception . . . 24
3.4 Junk-Byte-Insertion . . . 28
3.5 Java Class to load . . . 30
3.6 Dex Internal Storage . . . 31
3.7 Dex Method Invocation . . . 31
3.8 Self Modifying Code Example . . . 32

4.1 dlopen() Signature . . . 35
4.2 Internal Storage Initialization . . . 35
4.3 Buffered Input/Output . . . 36
4.4 Native libExe() . . . 37
4.5 Java Native Exec() . . . 39
4.6 C++ Native Exec() . . . 40
4.7 Reading /proc/self/maps . . . 42
4.8 machineCodeMul.c . . . 44
4.9 alloc_executable_memory() . . . 45
4.10 emit_code_into_memory() . . . 45
4.11 executeMachineCode() . . . 46
4.12 memoryWriting() . . . 48
4.13 crashApp() . . . 50
4.14 Egg Value Defining . . . 51
4.15 Egg Value and String Memory Sweep . . . 51
4.16 Native Code String Change . . . 53
4.17 NDK Encryption/Decryption . . . 54
4.18 Original vs Decompiled JNI Method Signatures . . . 56

A.1 MainActivity.java iii
A.2 MyNDK.java iii
A.3 mylib.cpp iii
A.4 Android.mk iv
A.5 Application.mk iv
A.6 build.gradle iv

B.1 AES Encrypt() v
B.2 AES Decrypt() vi
B.3 Memory/Android.mk vii
B.4 Decompiled AES Encrypt() vii

Bibliography

[ARM16a] ARM. *Development of TEE and Secure Monitor Code*. `http://www.arm.com/products/processors/technologies/trustzone/tee-smc.php`. 2016.

[ARM16b] ARM. *TrustZone*. `http://www.arm.com/products/processors/technologies/trustzone/`. 2016.

[Ash15] N. S. Ashutosh Jain Hugo Gonzalez. *Reverse Engineering through visual exploration of Android binaries*. Tech. rep. Banaras Hindu University, 2015.

[Ben13] E. Bendersky. *How to JIT - an introduction*. `http://eli.thegreenplace.net/2013/11/05/how-to-jit-an-introduction`. 2013.

[Ber15] P. Bernhard. "A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks." MA thesis. TU Munich, 2015.

[Bor08] D. Bornstein. *Dalvik VM Internals*. `https://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf?attredirects=0`. 2008.

[Col14] E. K. Collin Mulliner William Robertson. *VirtualSwindle: An Automated Attack Agains In-App Billing on Android*. Tech. rep. Northeastern University Boston, 2014.

[Com93] T. Committee. *Tool Interface Standard (TIS) - Portable Formats Specification*. Tech. rep. TIS Committee, 1993.

[Con09] J. Conrod. *Understanding Linux /proc/id/maps*. `http://stackoverflow.com/questions/1401359/understanding-linux-proc-id-maps`. 2009.

[Dom] M. P. Dominik Maier Tilo Müller. *Divide-and-Conquer: Why Android Malware*. Tech. rep. Friedrich Alexander Universität, -.

[Ele15] N. Elenkov. *Android Security Internals - An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2015.

[emi16] emileb. *OpenSSL for Android Prebuilt*. https://github.com/emileb/OpenSSL-for-Android-Prebuilt. 2016.

[Fal15] L. Falsina. *Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications'*. Tech. rep. Politecnico di Milano, 2015.

[Glo15] Gloryo. *How to create c++ library with NDK on Android Studio 1.5*. http://kn-gloryo.github.io/Build_NDK_AndroidStudio_detail/. 2015.

[Glo16] GlobalPlatform. *Trusted Execution Environment (TEE) Guide*. http://www.globalplatform.org/mediaguidetee.asp. 2016.

[Goo16a] Google. *Android Build Tool dx*. http://developer.android.com/tools/help/index.html. 2016.

[Goo16b] Google. *Android NDK*. http://developer.android.com/tools/sdk/ndk/index.html. 2016.

[Goo16c] Google. *Dalvik bytecode*. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. 2016.

[Goo16d] Google. *Dalvik Executable format*. https://source.android.com/devices/tech/dalvik/dex-format.html. 2016.

[Goo16e] Google. *DexFile Specification*. http://developer.android.com/reference/dalvik/system/DexFile.html. 2016.

[Goo16f] Google. *ProGuard*. http://developer.android.com/tools/help/proguard.html. 2016.

[Goo16g] Google. *Storage Options*. http://developer.android.com/guide/topics/data/data-storage.html. 2016.

[Gru16] B. Gruver. *Smali*. https://github.com/JesusFreke/smali. 2016.

[Gua16] GuardSquare. *DexGuard*. https://www.guardsquare.com/dexguard. 2016.

[Inc15] I. R. Inc. *Smartphone OS Market Share, 2015 Q2*. http://www.idc.com/prodserv/smartphone-os-market-share.jsp. 2015.

[Iso06] A. Isotton. *C++ dlopen mini HOWTO*. http://tldp.org/HOWTO/C++-dlopen/. 2006.

[Jan13] N. A. Jan-Erik Ekberg Kari Kostiainen. *Trusted Execution Environments on Mobile Devices*. Tech. rep. Trustsonic, 2013.

[JDr14] J. J.Drake. *Android Hacker's Handbook*. John Wiley and Sons, 2014.

[Jeo12] Y.-S. Jeong. *An Anti-Piracy Mechanism based on Class Separation and Dynamic Loading for Android Applications*. Tech. rep. Dankook University, 2012.

[Jon08] B. P. Jonathan M. McCune. *Flicker: An Execution Infrastructure for TCP Minimization*. Tech. rep. Carnegie Mellon University, 2008.

[Kal16] F. Kalhammer. *Das /proc-Dateisystem*. `http://www.linux-praxis.de/lpic1/lpi101/proc.html`. 2016.

[Ker15] M. Kerrisk. *POPEN(3) Linux Programmer's Manual*. `http://man7.org/linux/man-pages/man3/popen.3.html`. 2015.

[Kov12] X. Kovah. *Life Of Binaries Part 3*. `http://opensecuritytraining.info/LifeOfBinaries_files/2012_LifeOfBinaries3.pdf`. 2012.

[Kro16] J. Kroustek. *Retargetable Decompiler*. `https://retdec.com/home/`. 2016.

[Lev15] J. Levin. *Android Internals::A Confectioner's Cookbook - Volume I: The power users view*. Technologeeks.com, 2015.

[Mim15] G. Mimix. *memdlopen*. `https://github.com/m1m1x`. 2015.

[Mun14] M.-N. Muntean. "Improving License Verification in Android." MA thesis. TU Munich, 2014.

[Mur12] G. Murphy. *Position Independent Executables (PIE)*. `https://securityblog.redhat.com/2012/11/28/position-independent-executables-pie/`. 2012.

[Nat15] Y. S. Nathan Evans Azzedine Benameur. *All your Root Checks are Belong to Us:* tech. rep. Symantec Research Labs, 2015.

[Ora16] Oracle. *Java programming language compiler*. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html`. 2016.

[Pan16] Panxiaobo. *dex2jar*.

https://github.com/pxb1988/dex2jar. 2016.

[Sab15] P. Sabanal. *Hiding Behind ART*. Tech. rep. IBM Corporation, 2015.

[San12] E. J. Sanghoon Choi Joonhyouk Jang. *Android Application's Copyright Protection Technology based on Forensic Mark*. Tech. rep. Seoul National University Korea, 2012.

[Sch12] P. Schulz. *Code Protection in Android*. Tech. rep. Rheinische Friedrich-Wilhelms-Universität Bonn, 2012.

[Seb16] S. K. Sebastian Schrittwieser. *Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?* Tech. rep. St. Pölten University of Applied Sciences, Austria, 2016.

[Tim14] E. P. Tim Cooijmans Joeri de Ruiter. *Analysis of Secure Key Storage Solutions on Android*. Tech. rep. -, 2014.

[too15] tools.android.com. *Experimental Plugin User Guide*. http://tools.android.com/tech-docs/new-build-system/gradle-experimental. 2015.

[Unk16] Unknown. *Java Decompiler*. http://jd.benow.ca/. 2016.

[Wik16] Wiki.OpenSSL. *Android*. https://wiki.openssl.org/index.php/Android. 2016.

[Yag13] K. Yaghmour. *Embedded Android*. O'Reilly Media Inc., 2013.

[Yu14] R. Yu. *Android Packer*. https://www.virusbulletin.com/uploads/pdf/conference_slides/2014/Yu-VB2014.pdf. 2014.

[Yue14] H. Y. Yueqian Zhang Xiapu Luo. *DexHunter: Toward Extracting Hidden Code from Packed Android Applications*. Tech. rep. Hong Kong Polytechnic University, 2014.