



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Analysis of Android Cracking Tools and  
Investigations in Countermeasures for  
Developers**

Johannes Neutze, B. Sc.





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Analysis of Android Cracking Tools and Investigations in  
Countermeasures for Developers**

**Analyse von Android Crackingtools und Untersuchung  
geeigneter Gegenmaßnahmen für Entwickler**

Author:	Johannes Neutze, B. Sc.
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils Kannengießer, M. Sc.
Submission Date:	March 15, 2015



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

Munich, March 15, 2015

Johannes Neutze, B. Sc.

## Acknowledgments

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

## Assumptions

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

# Abstract

<http://users.ece.cmu.edu/~koopman/essays/abstract.html> Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Assumptions</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Glossary</b>	<b>1</b>
<b>Acronyms</b>	<b>2</b>
<b>1 Introduction</b>	<b>4</b>
1.1 Licensing . . . . .	4
1.2 Motivation . . . . .	4
1.3 Related Work . . . . .	5
<b>2 Foundation</b>	<b>6</b>
2.1 Software Piracy . . . . .	6
2.1.1 Developers . . . . .	6
2.1.2 Users . . . . .	8
2.1.3 Piracy on Android . . . . .	8
2.2 Android . . . . .	9
2.2.1 Android Application Package (APK) . . . . .	10
2.2.2 Dalvik Executable File Format . . . . .	12
2.2.3 Installing an APK . . . . .	13
2.2.4 Dalvik Virtual Machine . . . . .	14
2.2.5 Android Runtime . . . . .	16
2.2.6 Root and Copy Protection . . . . .	17
2.3 License Verification Libraries . . . . .	19
2.3.1 Google’s License Verification Library (LVL) . . . . .	19
2.3.2 Amazon DRM (Kiwi) . . . . .	23
2.3.3 Samsung DRM (Zirconia) . . . . .	25
2.3.4 Abstraction . . . . .	26

2.4	Code Analysis . . . . .	27
2.4.1	Retrieving an APK . . . . .	30
2.4.2	dex Analysis . . . . .	31
2.4.3	Smali Analysis . . . . .	32
2.4.4	Java Analysis . . . . .	34
2.4.5	Detect Code Manipulations . . . . .	36
<b>3</b>	<b>Cracking Android Applications with Lucky Patcher</b>	<b>38</b>
3.1	Lucky Patcher . . . . .	38
3.2	Code Analysis . . . . .	40
3.3	Analysis of Patched Applications . . . . .	41
3.4	Patching Patterns . . . . .	42
3.5	Conclusion and Learnings . . . . .	53
<b>4</b>	<b>Countermeasures for Developers</b>	<b>55</b>
4.1	Extend Current Library . . . . .	55
4.1.1	Modifications on the Google LVL . . . . .	55
4.1.2	Native Implementation of License Verification Library (LVL) . .	57
4.1.3	Tampering Protection . . . . .	58
4.1.4	Obfuscation . . . . .	64
4.2	Complex Verification . . . . .	68
4.2.1	Content Server . . . . .	68
4.2.2	Encryption . . . . .	69
4.3	Additional Environmental Features . . . . .	74
4.3.1	Trusted Execution Environment . . . . .	74
4.3.2	ART . . . . .	76
<b>5</b>	<b>Conclusion</b>	<b>77</b>
5.1	Summary . . . . .	77
5.2	Discussion . . . . .	77
5.3	Future Work . . . . .	78
	<b>List of Figures</b>	<b>79</b>
	<b>List of Tables</b>	<b>80</b>
	<b>List of Code Snippets</b>	<b>81</b>
	<b>Bibliography</b>	<b>83</b>



# Glossary

**.class** Java bytecode produced by the Java compiler from a .java file.

**.dex** Dalvik bytecode file, translated from the Java bytecode. Dalvik Executables are designed to run on system with memory or processor constraints. For example, the .dex file of the Phone application is inside the system/app/Phone.apk.

**.jar** The Java Archive is a package file containing Java class files and the associated metadata and resources of applications of the Java platform..

**.odex** Optimized Dalvik bytecode file are Dalvik Executables optimized for the current device the application is running on. For example, the .odex file of the Phone application is system/app/Phone.odex.

**ADB** The Android Debug Bridge is a command-line application providing different debugging tools.

**API** The Android Debug Bridge is a command-line application providing different debugging tools.

**APK** An Android Application Package is the file format used for distributing and installing applications on the Android operating system. It contains the applications assets, code (.dex file), manifest and resources.

**assembler** Ein Assembler (auch Assemblierer[1]) ist ein Computerprogramm, das Assemblersprache in Maschinensprache übersetzt, beispielsweise den Assemblersprachentext „CLI“ in den Maschinensprachentext „11111010“..

**disassembler** Ein Disassembler ist ein Computerprogramm, das die binär kodierte Maschinensprache eines ausführbaren Programmes in eine für Menschen lesbarere Assemblersprache umwandelt. Seine Funktionalität ist der eines Assemblers entgegengesetzt..

**Lucky Patcher** Android cracking tool used to remove license verification mechanisms from applications..

# Acronyms

**.dex** Dalvik EXecutable file.

**.jar** Java Archive.

**.odex** Optimized Dalvik EXecutable file.

**ADB** Android Debug Bridge.

**ADT** Android Developer Tools.

**AOT** Ahead-Of-Time.

**API** Application Programming Interface.

**APK** Android Application Package.

**ART** Android RunTime.

**DRM** Digital Rights Management.

**DVM** Dalvik Virtual Machine.

**ELF** Extensible Linking Format.

**GC** Garbage Collection.

**IP** Intellectual Property.

**JIT** Just-In-Time.

**JNI** Java Native Interface.

**JVM** Java Virtual Machine.

**LLVM** Low Level Virtual Machine.

**LVL** License Verification Library.

**NDK** Native Development Kit.

**OS** Operating System.

**OTG** USB On-The-Go.

**SDK** Software Development Kit.

**SE** Secure Element.

**TEE** Trusted Execution Environment.

**VM** Virtual Machine.

# 1 Introduction

## 1.1 Licensing

*Software Licensing* is the legally binding agreement between two parties regarding the purchase, installation and use of software according to its terms of use. It defines the rights of the licensor and the licensee. The goal is to protect the software creator's Intellectual Property (IP) or other features and enable him to commercialize it as a product. It defines the boundaries of usage for the user and prevents him from illicit usage [80].

Software licenses come in different variants. They range from open source over usage for a limited time to usage of a limited set of features. Since using the full feature set of software might be bound to paying a royalty fee, these software is often subject of piracy. In order to prevent unauthorized use, mechanisms to enforce the legal agreement are implemented. This includes Digital Right Management solutions which deny access to the software in case of a wrong serial key or unregistered account.

The problem is that these mechanisms do not offer absolute security and pirates always try to circumvent them. This results in an everlasting race of arms between software creators and software thieves [81].

## 1.2 Motivation

Licensing is also present in Android. With a market share of almost 82.8% in Q2 of 2015 [52] it is the most used mobile Operating System (OS). According to Google, this market share translates to over 1.4 billion active devices in the last 30 days in September 2015 [33]. This giant number of Android devices is powered by Google Play [50]. Google's marketplace offers different kinds of digital goods, as applications, music or movies, but also hardware. In the application section of Google Play user can choose from over 1.6 million applications for Android [85]. In 2014 Google's marketplace overtook Apple's Appstore, which had a revenue of over 10 billion back in 2013, and became the biggest application store on a mobile platform [58].

The growth comes with advantages. Some time ago developers only considered iOS as a profitable platform and thus most applications were developed for Apple's OS only or at least first. Now, with Android's overwhelming market share, they focus heavily on Android [68]. But this also creates a downside. The expanding market for Android, offering many high quality applications, draws the attention of software pirates. Crackers do not only bypass application's license mechanisms and offer them for free. Redirecting cash flows or distributing malware using plagiates is an lucrative business model as well.

Android developers are aware of the situation [88] and express their need to protect their IP on platforms like xda-developers [75] or stackoverflow [78]. Many of the developers have problems with the license verification mechanism and name *Lucky Patcher* as one of their biggest problems [79].

The scope of this thesis is to analyse Android cracking applications, like Lucky Patcher, and to investigate in countermeasures for developers.

### 1.3 Related Work

Lucky Patcher and license verification have already been topic of scientific work.

Patrick Bernhard takes a look at license verification an in-app billing attacks in his master's thesis *A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks*. He comes to the conclusion that the libraries need an overhaul since they are easy to circumvent and have not been updated for a long time. This shows the urgency on further investigation on this topic.

Marius-Nicolae Muntean's master's thesis *Improving License Verification in Android* does an analysis of techniques to crack Android's license verification and implements a new approach. He introduces multiple general strategies, such as obfuscation and dynamic code generation, to fortify code. In the end he uses the insights from the analysis to suggest countermeasures and their effects. A similar approach is chosen for this thesis.

## 2 Foundation

Before understanding the attack mechanisms and discussing countermeasures, necessary background knowledge has to be provided. Motivation and risks of software piracy and the basics of Android will be explained as well as existing licensing solutions. In addition, reengineering tools and methodologies for app analysis are described.

### 2.1 Software Piracy

According to Apple, 11 billion Dollars are lost each year due to piracy. Software piracy is defined as unauthorized reproduction, distribution and selling of software [28]. It includes the infringement of the terms of use of software by an individual as well as commercial resale of illegal software. Piracy is an issue on all platforms and is considered theft.

#### 2.1.1 Developers

Piracy is a big problem for developers as seen in figure 2.1. The developer loses direct revenue when his IP is stolen and redistributed by a pirate without the developer's involvement. In case the application is offered for free, users do not have to pay in order to download it and no revenue is created. It is even worse when the pirated application has to be purchased in another app store. In this case the pirate will get the profit which should be the developer's.

But revenue is not only lost when the application can be downloaded for free. The pirate is also able to dry up follow up revenue by modifying the application itself. There are two main types of indirect revenue. The first type are in-app purchases. They are a popular source of income for so called freemium applications or lite versions of applications. In case of the the freemium app, the download is for free and includes all features. The developer makes the money of in-app purchases like cosmetic modifications or in game currency. The lite version application is a little bit different. The download is free as well but the application comes with a restricted feature set or limited time of use. In order to take advantage of the full feature set the user can buy the pro license via an in-app purchase. Apps can include a mix or various degrees of theses types. Pirates can disable the transaction of the payments for the

in-app purchase. This makes the in-app purchases cost no money and thus no earnings are generated for the developer.

The second type of indirect revenue is generated by showing in-app advertisements. When this feature is implemented, advertisements are shown inside the application and the developer is paid by views and clicks on the advertisements. Earnings generated by an applications are assigned according to the included Ad Unit ID [49]. When an application is pirated, this ID can be replaced by the pirate's ID. Future revenues generated by advertisements will not be assigned to the developer but to the pirate.

Beside monetary issues, additional problems arise when the application is moved to a

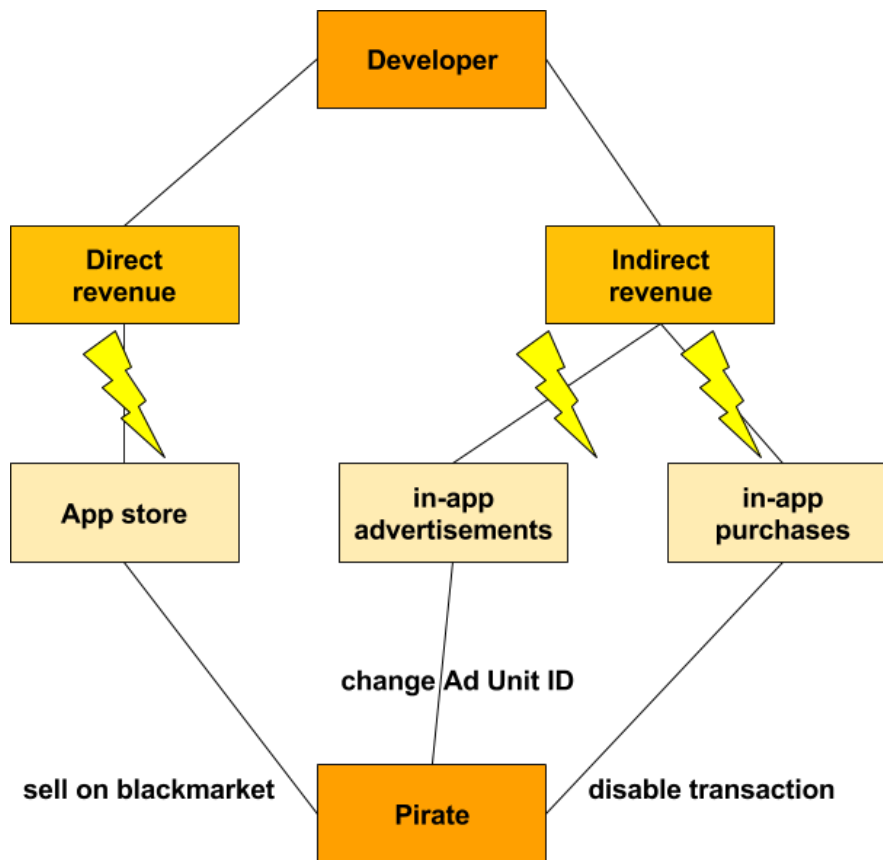


Figure 2.1: Different ways to generate revenue and how the pirate can cut them

black market store or website and distributed without the environment of an official app store. The results is the loss of control over the application for the developer. He can no longer provide support and updates for the application. The users will not get fixes for security issues and crashes caused by malfunctions. Users which do not know

that they are using a pirated version will connect the unsatisfying behaviour to the developer. This results in the loss of future revenues which are not even connected to this application.

The developer can face also unpredictable scenarios like unforeseen traffic since the growth of the application cannot be monitored with tools provided by the marketplace of choice. This can stress the server because they were not scaling accordingly. As revenue from the application is stolen, there may not be enough money for upgrades necessary by legal and illegal use [61].

Developers make a living of their applications. When they do not make a profit from their application, or even lose money with their servers, they can no longer continue. The result is a loss of creativity, ideas and skill for the ecosystem.

### 2.1.2 Users

The loss of developers in the ecosystem is bad for user, but they can also be harmed by piracy. Users use pirated applications because they seem to be free of charge, but it comes at a price. The application might be altered in different ways, e.g. malware may be included. The user will not notice it right away since these *features* often happen in the background without their knowledge. The application may for instance start using an expensive service, like premium SMS, or upload the contacts to the internet without the user recognizing. Even if there is no malicious content implemented, the application can suffer from bad stability due to manipulated code, like removed license verification, and missing updates when related from an unofficial source. In general, the risk is very high that pirated software has a worse user experience than the original. Pirated software should not be installed since the integrity of the application cannot be ensured without deep inspection. [32] [61]

### 2.1.3 Piracy on Android

Piracy is widespread on the Android platform. Especially in countries like China, piracy is as high as 90% due to restricted access to Google Play [43]. Sources for pirated applications can be easily found on the internet. A simple search, containing *free apk* and the applications name, returns plenty of results on Google Search. The links direct to black market applications, as Blackmart [30], and websites offering cracked Android Application Package (APK), such as crackApk [40]. They claim to be user friendly because they offer older versions of applications. Their catalog even includes premium apps, which are not free in the Play Store and include license verification mechanisms [27]. Offering these applications is only possible when the license mechanism is removed. They practice professional theft and expose users to threats (see section 2.1.2).



*Today Calendar Pro* is an example for the dimensions piracy can reach for a single application. The developer stated in a Google+ post that the piracy rate of the application is as high as 85% at the given day. [75] [88] Since it looks like that license mechanisms are no obstacle for pirates and sometimes cracked within days, some developers do not implement any copy protection [54].

Android applications are at an especially high risk for piracy because bytecode in general is an easy target to reverse engineer as shown in the further proceeding.

## 2.2 Android

Android is an open source mobile OS launched in 2007 and today mainly maintained and developed by Google. It is based on the Linux kernel and mainly targets touch screen devices such as mobile devices or wearables. The system is designed to run efficiently on battery powered devices with limited hardware and computational capacity. Android's main hardware platform is the ARM architecture, known for their low power consumption. The following will give an overview over the architecture of Android and later a deeper insight in the runtime system powering Android. The architecture of the software stack of Android can be seen in figure 2.2.

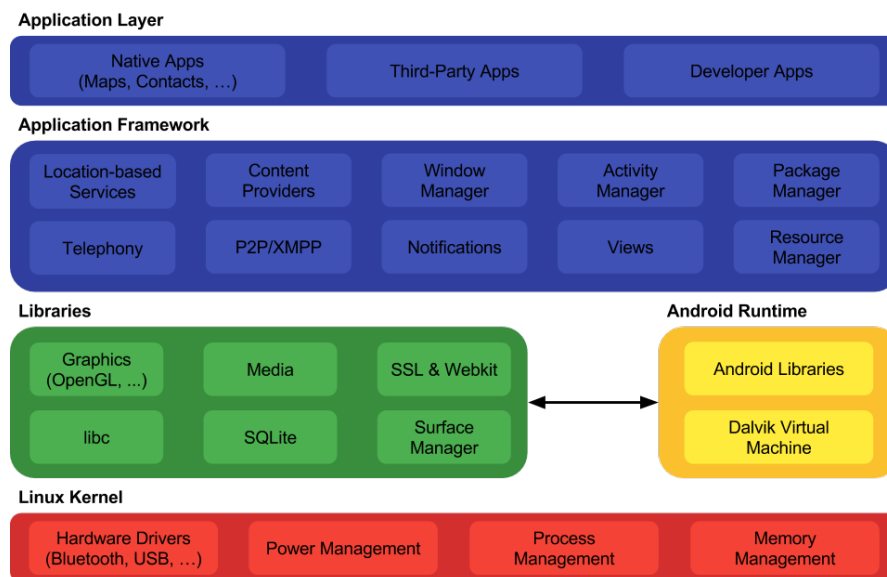


Figure 2.2: Android's architecture [65]

The basis of the system is its kernel. It is responsible for power and memory man-

agement and controls the device drivers.

The layer above the kernel contains the Android RunTime (ART) as well as the native libraries of the system. ART and its predecessor, the Dalvik Virtual Machine (DVM), will be covered in section 2.2.4 and in section 2.2.5. Usually, Android libraries are written in Java, except the ones which are resource and time critical. They are written in C or C++ in order to boost performance and allow low level interaction between applications and the kernel by using the Java Native Interface (JNI). Examples for native libraries are OpenGL, multimedia playback or the SQLite database.

On top of the libraries and the runtime lies the application framework. This layer provides generic functionality, such as notification support, to applications over Android's Application Programming Interface (API).

The top layer enables the installation and execution of applications.

Using these layers and abstraction allows software to execute standard Unix commands on the kernel and Android to run on a wide range of devices with different hardware and.

### 2.2.1 Android Application Package (APK)

Android applications are distributed and installed using the APK file format. They can either be obtained from an application store, like Google Play, or downloaded and installed, manually or by using Android Debug Bridge (ADB), from any other source. The APK format is based on the ZIP file archive format and contains the code and resources of the application.

The build process of APK contains several steps which are visualized in figure 2.3.

Since Android applications are usually written in Java, the start is similar to the Java program build process. Android applications are usually written in Java. They have the same build process as standard Java applications. Upon compilation, the source code is compiled to .class files by the Java Compiler `javac`. Each Java class is stored as bytecode in the corresponding .class file. Java bytecode can be obfuscated, which is topic of section 4.1.4. When all Java classes are compiled to .class files, they are packed into a Java Archive (.jar) file.

Since Android is using a different Virtual Machine (VM) for executing the code, the Java bytecode has to be converted to Dalvik bytecode. The Android Software Development Kit (SDK) provides `dx`, the tool used to convert .class files to a single classes.dex file. The VM and the Dalvik EXecutable (.dex) file format will be described in the following. Similar to the Java bytecode, obfuscation can be applied.

The APK itself consists of three parts.

- classes.dex, containing the bytecode

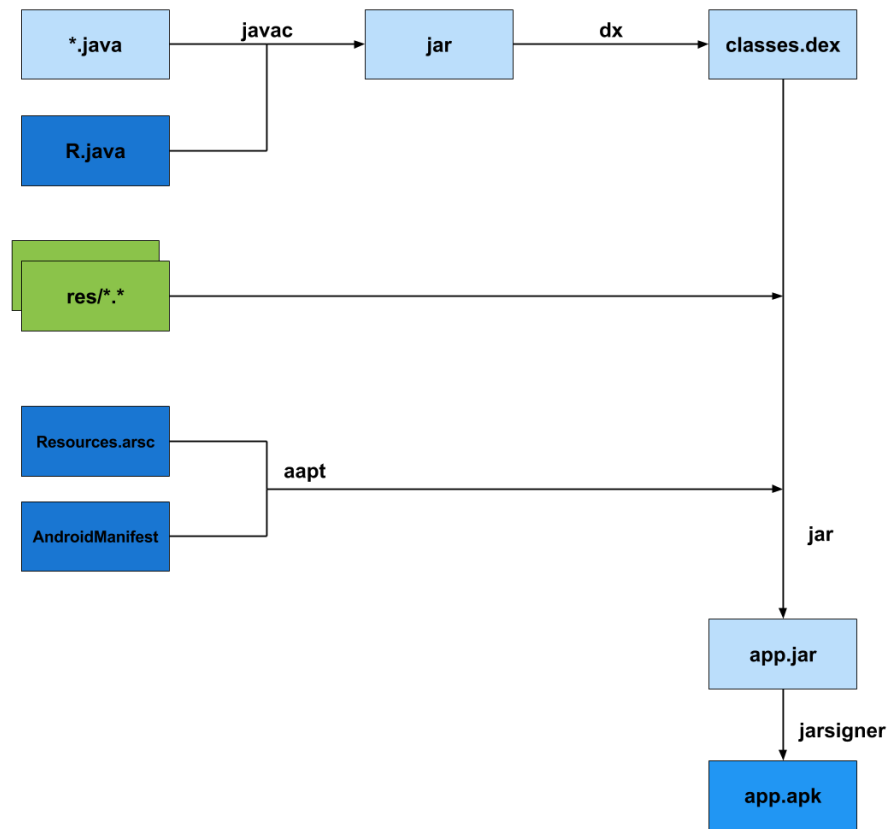


Figure 2.3: APK build process [60]

- resource files (`res/*. *`), containing static content like images, the `strings.xml` and the `layout.xml` files
- `resources.arsc` and `AndroidManifest.xml`, containing compiled resources respectively essential information as required permissions

The *apkBuilder* combines these files into one archive file.

Before releasing the application, it has to be signed and zipaligned. The *jarsigner* is used to sign the application with the private key of the developer. It enables Android to identify the developer and support future updates for the application. Afterwards *zipalign* is used to mark uncompressed data. [23] [60]

The structure of a final APK file has at least the following content seen in figure 2.4. The `AndroidManifest.xml` and the `classes.dex`, which have been covered already. The `META-INF` folder, which is inherited from Java and used to store package and extension

```
|-- AndroidManifest.xml
|-- META-INF
|   |-- CERT.RSA
|   |-- CERT.SF
|   `-- MANIFEST.MF
|-- classes.dex
|-- res
|   |-- drawable
|   |   |-- icon.png
|   |   |-- layout
|   |   `-- main.xml
|   `-- resources.arsc
```

Figure 2.4: APK folder structure

configuration data, e.g. the signature [66]. While the static resources, like drawables and layouts, are in the res folder, the resources.arsc contains the compiled resources. In case the application implements native code, it is stored in the libs folder, split by the different processor types, like armeabi-v7a for ARM or x86 for Intel processors. [56] [44]

### 2.2.2 Dalvik Executable File Format

As explained in chapter [subsection:foundation-android-package], Android applications are distributed using dex bytecode which is compiled from Java bytecode. Dalvik bytecode is suited to run on the ARM architecture. It supports direct mapping from dex registers to the 32 bit registers of the ARM processor. The instructions are 16 bit multiples, which makes the dex bytecode less compact than Java bytecode with its 8 bit instructions. The 16 bit instructions result in 218 valid opcodes which have a dest-source ordering for its arguments. [11] In case there are 64 bit values, adjacent registers are used to store it.

Similar to Java bytecode, instructions are not stored inside the method but as a reference pointing to the variable. While in Java each class has its constants, like numbers, strings and identifier names, grouped together in heterogenous pool (see figure figure 2.5, left side), Dalvik bytecode uses one pool for each type. When compiling Java bytecode to Dalvik bytecode, the heterogenous pools of each Java class are merged together in one global pool for each type (see figure figure 2.5, right side). In this process, duplicates in a pool are removed, which reduces memory need for constant but increases the number of references. This is most effective for strings. A decrease of the memory footprint of

up to 44% compared to the .jar is possible.

The compiled .dex file has the the structure seen in figure 2.6 on the left. Most

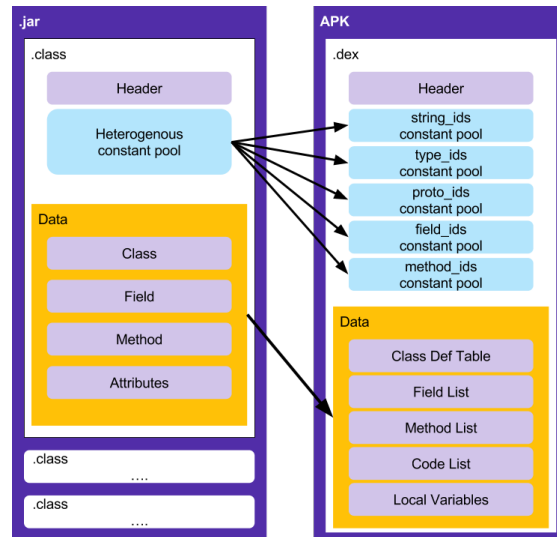


Figure 2.5: .jar to APK transformation [31]

important parts of the header are the checksum and the signature. The checksum contains the Adler32 checksum of the .dex file, including everything except the magic and this field. It is used to detect whether the file is corrupt. The signature contains the SHA-1 signature of the content of the file, except the magic, checksum and this field. The field is used to uniquely identify the file. When the file is modified, both values have to be updated. [17] [44]

Dex bytecode supports optimization, upon installation improvements for the underlying architecture can be applied to the bytecode. The resulting .dex file is called Optimized Dalvik EXecutable (.odex). The optimization is executed by a program called *dexopt* which is part of the Android platform. The semantics of the two files is the same, but the .odex file has the better performance.

Like Java bytecode, .dex bytecode has a serious flaw. Since bytecode is pretty simple and contains a lot of meta information, decompilation can be succesfully done and the result is easily understandable. At the same time, protection is rarely applied by the developers. This makes these applications an easy target for reverse engineering.

### 2.2.3 Installing an APK

Before running an application, it has to be installed. The installation consists of two major steps. The first step is primarily about verification, while the second step is the

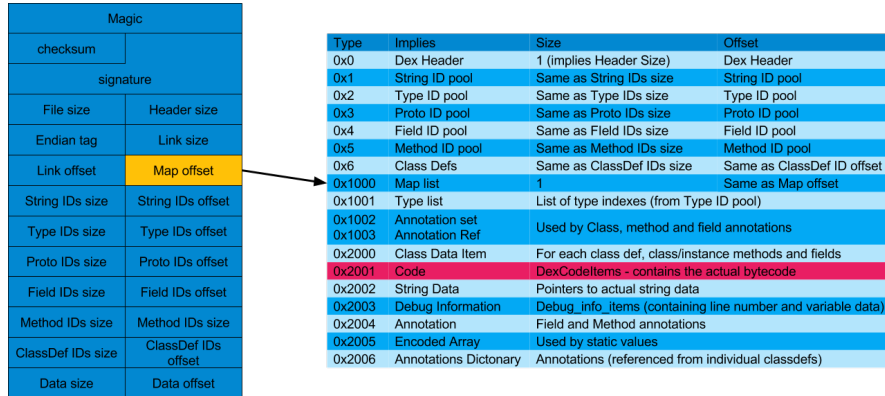


Figure 2.6: .dex file format [60]

bytecode optimization and, in case of ART, the code compilation. The differences and procedures of the DVM as well as ART are explained in detail in the following chapters (see figure 2.7). Before initiating the installation, the APK is checked for a legitimate signature and checksum as well as correct classes.dex structure.

In case this cannot be verified, the installation is rejected by the OS. The second step is the optimization where the .odex version of the classes.dex is generated. As a reminder, the .odex is an optimization tailored to the specific architecture of the device in order to achieve the best performance. This is useful due to the the high diversity of Android running hardware and their different processors. When using the DVM, the classes.dex file is taken from the archive and put as .odex file into the Dalvik cache. This is done once and from then on the execution is done by using the .odex file. This preprocessed version of the application has an improved startup time. [56]

Currently, the Android runtime of choice is ART. For this runtime, the second step is more complex since the bytecode has to be compiled an additional time. This will be explained closer in section 2.2.5.

After the bytecode is optimized respectively compiled, the application can be run. Android is creating an sandboxed environment for each application when run. This is achieved by Android assigning each process a seperate user ID on install to ensure that each application is isolated from the others and has no access on resources except its own. [15]

## 2.2.4 Dalvik Virtual Machine

The original VM powering Android is the DVM, which was created by Dan Bornstein and named after an Icelandic town. It was introduced along with Android in 2008 [24].

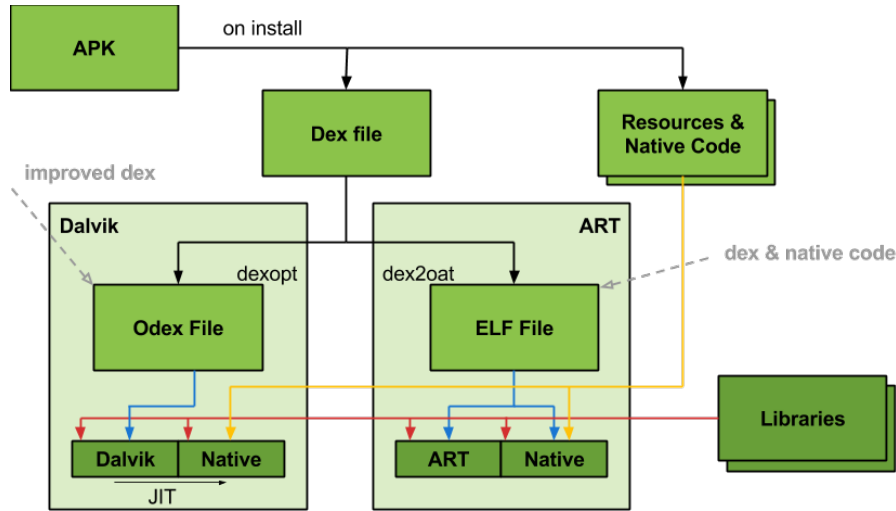


Figure 2.7: Installing an APK on a device [26]

In contrast to a stationary computer, a mobile device has a lot of constraints. Since it is powered by battery the processing power and RAM are limited to fit power consumption. In addition to these hardware limitations Android has some additional requirements, like no swap for the RAM, the need to run on a diverse set of devices and in a sandboxed application runtime. In order to deliver best performance and run efficiently, it has to be designed according to these requirements. The DVM is a customized and optimized version of the Java Virtual Machine (JVM) and based on Apache Harmony. Even though it is based on Java, it is not fully J2SE or J2ME compatible since it uses 16bit opcodes and register-based architecture in contrast to the standard JVM being stack-based and using 8bit opcodes. The advantage of register-based architecture is that it needs less instructions for execution than stack-based architecture which results in less CPU cycles and thus less power consumption which is important for a battery driven device. The downside of this architecture is the fact that it has an approximately 25% larger codebase for the same application and negligible larger fetching times [44]. In addition to the lower level changes the DVM is optimized for memory sharing, it stores references bitmaps separately from objects and optimizes application startup by using zygotes [60].

The last big change made to the DVM was the introduction of Just-In-Time (JIT), which will be discussed in subsection 2.2.5, in Android version 2.2 "Froyo".

### 2.2.5 Android Runtime

<!-- benötigt noch feinschliff -->

In Android version 4.4 *Kitkat* Google introduced ART which was optional and only available as a preview through the developer options. Like DVM, ART works according to the dex bytecode specification and executes the .dex fileformat [9]. With the release of version 5.0 *Lollipop* ART it became the runtime of choice since DVM had some major flaws. Throughout the Android 6.0 *Marshmallow* previews it was constantly evolving and sometimes breaking with older versions at the cost of almost no documentation. ART is designed to address the shortcomings of the DVM. Maintaining an VM is expensive, having an interpreter and JIT is not as efficient of native code and doing JIT each time the application is executed is wasteful. This and maintaining background threads require significantly more CPU cycles which can be directly translated to slower performance and increased battery usage. In addition the Dalvik Garbage Collection (GC) frequently causes hangs and jitters and DVM is only supporting 32 bit. With ART Android is following iOS into the 64 bit world but this is not the only advantage over the DVM. Improvements in the VM make the maintenance less expensive and the compilation is changed from JIT to Ahead-Of-Time (AOT) as well as the overhead cycles have been reduced. The GC is also non-blocking now and can run parallel in fore- and background.

The main idea of ART and AOT is to compile the application to one of two types, either native code or Low Level Virtual Machine (LLVM) code. Each of the types has its purpose and advantage. The native code offers an improved execution performance while the LLVM code offers portability. In practice the preference is to compile to native since adding LLVM bitcode adds another layer of complexity to ART.

Different from DVM is the fact that ART uses not one but two file formats. Similar to the zygote of DVM, ART offers an image of pre-initialized classes and related object at run time, the boot.art file. It is mapped to the memory upon zygote startup to provide improved application starting time [10]. The second file format is

Art itself: art uses not one but two file formats art - only one file, boot.art, in /system/framework/architecture (arm,...) oat - master file, boot.oat, in /system/framework/architecture (arm,...) - odex files no longer optimized dex but oat, alongside apk for system apps/frameworks, /data/dalvik-cache for 3rd party apps, still uses odex extension, now file format is elf/oat

art files is a proprietary format, poorly documented, changed format internally repeatedly art file maps in memory right before oat, which links with it contains pre-initialized classes, objects and support structures

ART/OAT files are created (on device or on host) by dex2oat art still optimizes dex but



uses dex2oat instead, odex files are actually oat files (elf shared objects WAS IST DAS), actual dex payload buried deep inside command line saved inside oat file's key value store

finding dex in oat odex files will usually have only one (=original) dex embedded booat.oat is something else entirely, some 14 dex files the best of the android framework jars, each dex contains potentially hundreds of classes

art supports multiple architectures(x86,arm/64,mips) compiler is layered architecture, using portable (llvm) adds another lvl with llvm bitcode (not in this scope)

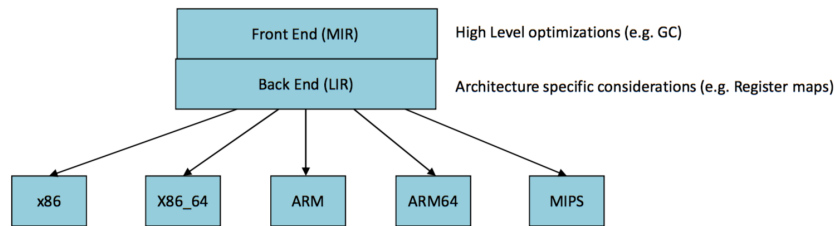


Figure 2.8: artarch

lessons base code is dex so vm is still 32 bit, no 64 bit registers or operands so mapping to underlying arch inst always 64 bit, there are actually a few 64 bit instructions but most dex code doesn't use them generated code isn't always that efficient, not on same par as an optimizing native code compiler, likely to improve with llvm optimizations overall code (determined by Mir optimizations) flow is same garbage collection, register maps, likewise same caveats: not all methods guaranteed to be compiled, reversing can be quite a pain

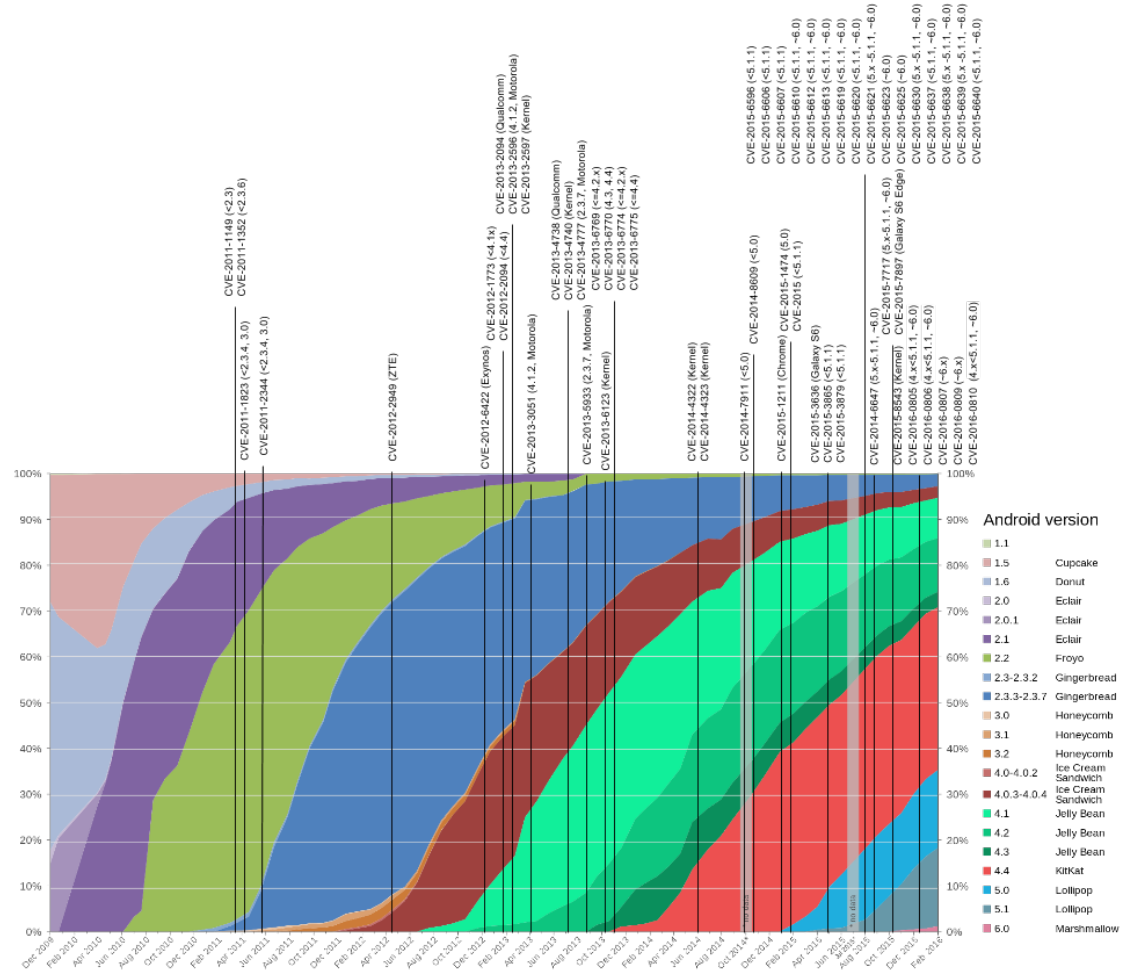
isn't android all dalvik now? art is runtime but application compile into dex, art is compiled on device during install, art binaries has dalvik embedded, some methods may be left as dex to be interpreted, dalvik is much easier to debug than art –see-evaluation

[60] When creating odex on art it is directly put into art file

## 2.2.6 Root and Copy Protection

*Rooting* or *getting root* is the process of modifying the operation system's software that shipped with a device in order to get complete control over it. The name *root* comes from the Linux OS world where the user *root* has all privileges. This allows to

Today it is easy to exploit these vulnerabilities and gain root rights, even for non-



techies. There are videos and tutorials available on the internet, even tools to automate the process, like Wugfresh's Rootkits [89]. Rooting is usually bundled with installing a program called *su* which manages the root access for applications requesting it. Rooting a phone is not without risk since installing bad files can result in the so called *bricking* meaning the phone is nonfunctional since the software cannot be executed anymore. [63]

Now that the application is installed and ready to run. Copy protection is applied to prevent unauthorized usage of the app. The downloaded APK, purchased from an application store, is moved to `/mnt/app-asec/package.name` on the phone. The user has no rights to access the containing APK and thus cannot copy it. This mechanism has a major flaw, as copy protection is circumvented when a single user can get hold of the application and redistribute it, e.g. by using root. This was effective in the early days of Android when rooting was not easily facilitated.

Since rooting circumvents the copy protection, it is declared as deprecated and new ways of protecting applications have to be invented.

## 2.3 License Verification Libraries

Since the original approach of subsection 2.2.6 was voided, another method had to be introduced. This topic is not only relevant to Google as the main contributor to Android and provider of its biggest store. Since Android allows to install apps from unknown sources and not only the Google Play, other stores were created to get a piece of Google's Android pie. Some of the most widespread stores are from Amazon and Samsung. Amazon does not only have the Amazon Store but is also trying to create an own ecosystem by selling the Firetablets with an Amazon tailored flavor of Android at a low price tag. Another approach is Samsung pursuing. In addition to a store they are also offering different services as well to bind to their ecosystem. There are different Chinese stores as well but they are out of scope since their relevance is bound to the eastern markets.

All these stores have to fight the copy protection problem in order make their store attractive and bind developers with low piracy rates. Since the initiative is coming from the stores, the copy protection methods are included into application itself. [64]

### 2.3.1 Google's License Verification Library (LVL)

In order to tackle the copy protection problematic and to give the developer community a possibility to fight piracy, Google introduced the LVL at 07/27/2010 [37]. It is simple and free of charges. The documentation can be found on the Android developers website. [19]

### Functional Principle

Google's approach is based on a network service which allows to query the trusted Google Play license server in order to determine whether the user has a valid license. The LVL is provided by Google in source code to be included in an App by the programmer. Thus its exact functionality and even structures and variable names can be studied by programmers. [64]

It has to be manually integrated into the application by the developer and allows a simple checking with Google and has a callback for the reply. The developer can decide when and how often the application should check its license. Upon initialization, the library connects to the Google Play Service which manages the connection between the device and the license server. It sends a request for a license check to the server. This is similar to Digital Rights Management (DRM). Adding the library to the application does not alter the function of the application, it just adds a call. The developer only needs to handle the result of this call, as the library takes care of the complicated process, like the networking and web service, and returns the response of Google Play as soon as it arrives. The developer is in full control of what happens with the response and whether access is granted. Figure 2.10 gives an overview how the parts are connected. [55] [19]

In order to start the validation process with the Google Server and determine the license status of the application, additional information has to be provided. The *LicenseChecker* has to be initiated with the package name of the application, a nonce, to ensure integrity of the response, as well as the callback for asynchronous handling, to start the call to the Play Service client. The Google Play Service then adds the primary Google account username, the identity and other information and sends the license check request to the server. On the Google Play server, it is checked whether the user has purchased the application and a corresponding response is send back to the Google Play Service, which returns it to the application. [19]

The security of the response is very important. It is achieved by public/private key encryption. Each published application in the Play Store has a pair of keys. The developer has to integrate the public key into the application, which is available in the Google Play developer console. The private key is used to decrypt the communication with the server which encrypts its response with the private key. This way it the origin of the response is ensured and tampering detected. [64] [19]

This mechanism requires a connection to the internet, as it needs to connect to the Google server. In case this is not possible, an internal error will be triggered which results in the license not being verified. In addition, the Google Play Service has to be installed which comes preinstalled with the Google Apps, including the Google Play store. It is a prerequisite to legally acquire an application with the LVL since it

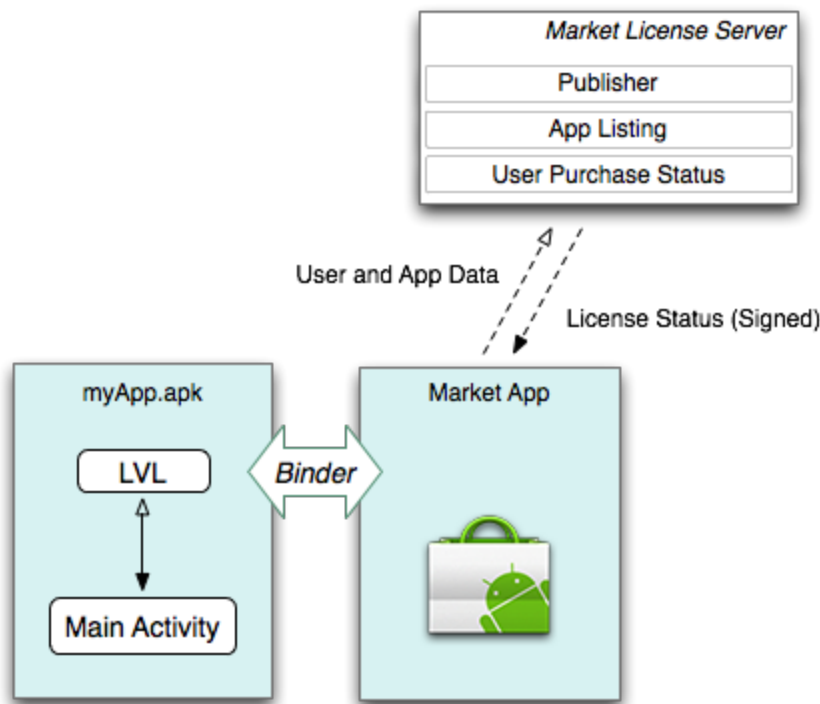


Figure 2.10: Google's implementation of license checking [19]

is bought in the Play Store. In case the application is installed and the Google Play Service is not installed, it cannot bind to it and thus cannot verify the license. The developer can decide when and how often the license check should be done as well as whether the result should be stored for future requests or when no internet connection is available. [**developersLicensingAdding**] [19]

The LVL mechanism replaces the old copy protection, which is no longer supported. This license verification model can be enforced on all devices which have access to Google Play. [13] [19]

### Implementation of the License Verification Library

The provided library of Google can be implemented in only a few steps. First of all, a Google Publisher Account for Google Play is needed. It enables the developer to publish applications on Google Play and take advantage of the LVL. As soon as an application is created in the Google Developer Console the public/private key is created. Each key pair is app specific and can be found under "Services & API". It is later on added into the application. [21]

Now that the Google Play prerequisites are set, the LVL has to be extracted from the Android SDK folder and added to the application. As soon as the LVL is part of the application project the licensing permission can be added to the application's manifest (see code snippet 2.1), else the LVL will throw an exception. The last step is to create

```
7    ...
8    <uses-permission android:name="com.android.vending.CHECK_LICENSE" />
9    ...
```

Code Snippet 2.1: Include permission to check the license in AndroidManifest.xml [13]

the license call which can be seen in code snippet 2.2. As described in subsection 2.3.1,

```
57    final String mAndroidId = Settings.Secure.getString(this.getContentResolverSettings.Secure.
58        ANDROID_ID);
59    final AESObfuscator mObfuscator = new AESObfuscator(SALT, getPackageName(),
60        mAndroidId);
61    final ServerManagedPolicy serverPolicy = new ServerManagedPolicy(this, mObfuscator);
62    mLicenseCheckerCallback = new MyLicenseCheckerCallback();
63    mChecker = new LicenseChecker(this, serverPolicy, BASE64_PUBLIC_KEY);
64    mChecker.checkAccess(mLicenseCheckerCallback);
```

Code Snippet 2.2: Setting up the LVL license check call

different information has to be acquired and added to the license request. In order to identify the user the its unique id is acquired. The id is created randomly when the user sets up the device for the first time and remains the same for the lifetime of the user's device [22]. The obfuscator is later applied on the cached license response data to prevent manipulation or reuse by root user. In order to decide what should happen to the response data and whether it should be stored, a policy is specified. This can be either one of the provided policies or a custom one. Before making the license check request, a callback has to be defined, which can be seen in code snippet 2.3. The callback contains actions for the different outcomes of the license check. The scenarios are either applicationError, e.g. when no connection can be established, dontAllow in case license is not valid and allow when the user's status is verified. Now the license check can be initiated with the values. [21][13]

The license check can be implemented in any activity and executed as the developer likes.

```
57
58     @Override
59     public void allow(final int reason) {
60         ...
61     }
62
63     @Override
64     public void dontAllow(final int reason) {
65         ...
66     }
67
68     @Override
69     public void applicationError(final int errorCode) {
70         ...
71     }
72 }
73 }
```

Code Snippet 2.3: LVL license check callback

### 2.3.2 Amazon DRM (Kiwi)

As an alternative to Google Play, started its own application store in October 2010 with the goal to generate profits from selling apps on Android as well [7]. It was opened to the public on the 03/22/2011 for Android and Fire tablets [8]. The store comes with its own DRM since the Google LVL only works with the Google Play Store. The developer can chose whether it should be enabled in the developer console in their developer console. When reengineering the APK the added code can be found in a package called Kiwi as seen in figure 2.11. [6]

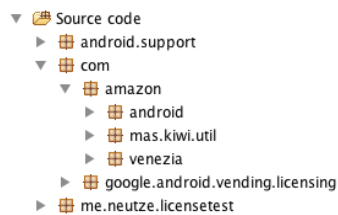


Figure 2.11: Amazon library structure in decompiled application

## Functional Principle

The prerequisites are the same as with the LVL, the developer has to have an account on the Amazon platform. Amazon has a different approach of implementing the license verification check. Instead of the developer integrating the license verification library himself, the developer is asked when uploading the application whether it should be integrated (see figure 2.12). According to the description, the library is to *Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.* [6] In order to implement the library the APK is decompiled server sided, the library is added and the package is then signed with a new signature (see subsection 2.2.1). Amazon describes it as following: *As part of the ingestion process Amazon removes your developer signature and applies an Amazon signature. This signature is unique to you, does not change, and is the same for all apps in your account.* [6]

**Apply Amazon DRM? \***  
Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.

☒ Yes (Recommended) ☐ No

**Appstore Certificate Hashes**  
As part of the ingestion process Amazon removes your developer signature and applies an Amazon signature. This signature is unique to you, does not change, and is the same for all apps in your account.

SHA-1 ⓘ	Hexadecimal	53:A8:F2:16:61:15:B0:D8:3B:2E:D2:BC:9B:80:7B:F7:64:F6:E3:2C
	Base64	U6jyFmEVsNg7LtK8m4B792T24yw=
MD5 ⓘ	Hexadecimal	F8:C6:B6:83:39:5F:85:AA:D3:D2:BF:84:74:C7:D9:9C
	Base64	+Ma2gzlffharT0r+EdMfZnA==

Figure 2.12: Developer preferences in the Amazon developer console [6]

amazon appstore has to be installed the whole time and user has to be logged in order that the DRM works

airplane and first time activity, callback: license not verified second time stored license

## Implementation of Kiwi

kind of wrapper no sample implementation to add by developer but inject own logic in each app (same for every app)

example shows implementation recovered by reengineering explained in 2.4 amazon drm contains numerous namespaces and calsses, most have been mangled by obfuscation tools, see proguard startup in main activity myActivity drm logic not interweaved with app logic, could only be done by a human developer anyways[64]

rename onCreate to onCreateMainActivity start in new onCreate, also start Kiwi.onCreate((Activity) this, true); which handles the



```

77 public void onCreate(Bundle bundle) {
78     onCreateMainActivity(bundle);
79     Kiwi.onCreate((Activity) this, true);
80 }

```

Code Snippet 2.4: Amazon's onCreate() injection to call Kiwi license verification as well

### 2.3.3 Samsung DRM (Zirconia)

Samsung as a major player in the smartphone business has also his own app store [38] Galaxy Apps by Samsung, formerly known as Samsung Apps, for devices by Samsung renamed in July 2015 called zirconia for android [72]

#### Functional Principle

library providing preventive measure against illegal reproduction works only on Samsung devices since Samsung Store has to be installed and logged in inspects the license of application executed to prevent illegal use checks for license from license server upon init and stores it on device for future offline check, timed life also checks if license from server if stored license has been removed or damaged, license from server unique for each device/application if app is copied to another device, application will not execute

interior process: makes query for stored licensetest if found, app can execute if not exist or invalid, information of device and application will be sent to server (once stored internet connection not required anymore) if purchased for device, server returns license back to zirconia zirconia stores license on device return step 1

callback method, asynch, ??zirconia does not return license validity result as boolean ??2.6 does not work if network is offline or in airplane

[72]

no store services needed, direct communication with server

airplane first time activity, callback: license not verified second time verified if license was stored

#### Implementation of Zirconia

java package .jar and JNI native library have to be added to project for check and query of license server zirconia needs device info and internet connection (READ\_PHONE\_STATE and INTERNET permission)

4 basic steps (see 2.5) create

can be implemented in any stage of the application, e.g. start or when saving

[72]

```
57     final Zirconia zirconia = new Zirconia(this);
58     final MyLicenseCheckListener listener = new MyLicenseCheckListener();
59     listener.mHandler = mHandler;
60     listener.mTextView = mStatusText;
61     zirconia.setLicenseCheckListener(listener);
62     zirconia.checkLicense(false, false);
```

Code Snippet 2.5: Setting up the Zirconia license check call

```
57     @Override
58     public void licenseCheckedAsValid() {
59         mHandler.post(new Runnable() {
60             public void run() {
61                 ...
62             }
63         });
64     }
65
66     @Override
67     public void licenseCheckedAsInvalid() {
68         mHandler.post(new Runnable() {
69             public void run() {
70                 ...
71             }
72         });
73     }
```

Code Snippet 2.6: Zirconia license check callback

First looks great, puts the copy protection inside the app, a from of DRM  
communicate with server, authorize use of application  
does not prevent user from copying/transferring app, but copy useless since the app  
does run without the correct account  
google die ersten, andere folgen, anfangs problem, dass dadurch nur durch google  
store geschützt war, grund dafür dass evtl ein programmierer in meinen store kommt

### 2.3.4 Abstraction

basiert einzig auf der antwort vom server die ja oder nein ist

but there are problems as well android's content protection are invalid when rooted  
DRm can be bypassed coupled with dex decompilkation big problem, app can be

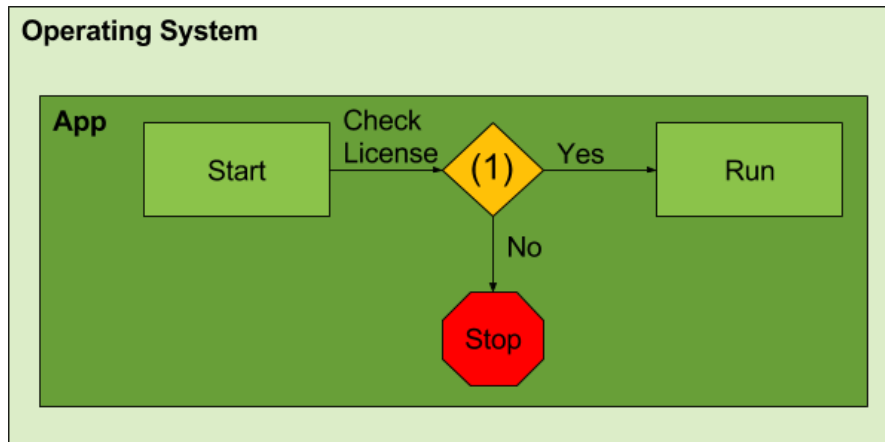


Figure 2.13: Abstraction of the current license verification mechanism. The library is represented by (1)

decompiled, modded and repackaged[59]

Es muss generell immer abgewogen werden zwischen Reichweite und Sicherheit. Von Output den Lucky Patcher gibt, sind die auto patching modes für Google, Amazon und Samsung, die großen Player. Ein Developer muss seine App dort anbieten um Aufmerksamkeit zu bekommen. Deswegen sind diese Stores auch so gut "maintained" von Lucky Patcher. Im Falle, dass ein Developer "Sicherheit" vorzieht und seine App in einem alternativen Store anbietet, gibt es zwei Szenarien. Entweder entwickelt jemand einen Custom Patch (dex oder native Angriff) wenn ein "allgemeines Interesse" besteht oder die App ist uninteressant und erhält keine Aufmerksamkeit, weder von LP noch Kunden.

## 2.4 Code Analysis

The Cracking Tool has to alter an application's behaviour by applying patches only to the APK file, since it is the only source of code on the phone. This is the reason for the investigations to start with analysing the APK. This is done using static analysis tools. The aim is to get an accurate overview of how the circumventing of the license verification mechanism is achieved. This knowledge is later used to find countermeasures to prevent the specific cracking tool from succeeding.

The reengineering has to be done by using different layers of abstraction. The first reason is because it is very difficult to conclude from the altered bytecode, which is not human-readable, to the new behaviour of the application. The second reason is because the changes in the Java code are interpreted by the decompiler, which might not reflect

the exact behaviour of the code or even worse, cannot be translated at all. These problems are encountered by analysing the different abstraction levels of code as well as different decompilers.

recover the original code of an application bytecode analysis is most often used. By applying both dynamic and static techniques to detect how behavior is altered dynamic analysis during runtime, static raw code, done by automatic tools using reverse engineering algorithms, best case whole code recovered, worst case none

When speaking of reverse engineering an Android application we mostly mean to reverse engineer the bytecode located in the dex file of this application.

The classes.dex file is a crucial component regarding the application's code security because a reverse engineering attempt is considered successful when the targeted source code has been recovered from the bytecode analysis. Hence studying the DEX file format together with the Dalvik opcode structure is tightly related to both designing a powerful obfuscation technique or an efficient bytecode analysis tool. [56]

dex to java .dex and .class are isomorphic dex debug items map dex offsets to java line numbers tools like dex2jar can easily decompile from dex to a jar extremely useful for reverse engineering, even more so useful for malice

flow from dex to java is bidirectional, decompile code back to java, remove annoyances

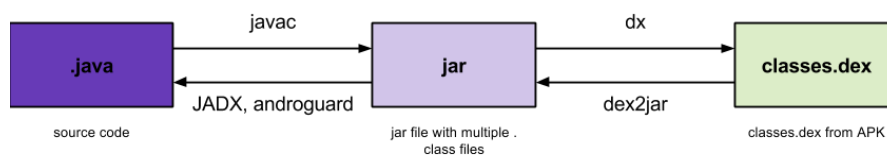


Figure 2.14: Java .class and .dex can be transformed bidirectional [60]

like ads, registration, uncover sensitive data (app logic, secrets), replace certain classes with others (malicious ones), recompile back to jar, then dex, put cloned/trojaned version of your app on play or other market [60]

android vulnerability of app is reverse engineering the source code, patching security mechanism and recompiling the app best case scenario is obtaining one to one copy of original source code since reading and understanding high level code is easiest so will the patching be reality often not possible due various protecting of source code, also unnecessary since lower level representation of source code might be enough to reveal mechanism patch and compile low level code tools and documentation have matured many tools and techniques

gaining information about a program and its implementation details, process aims at

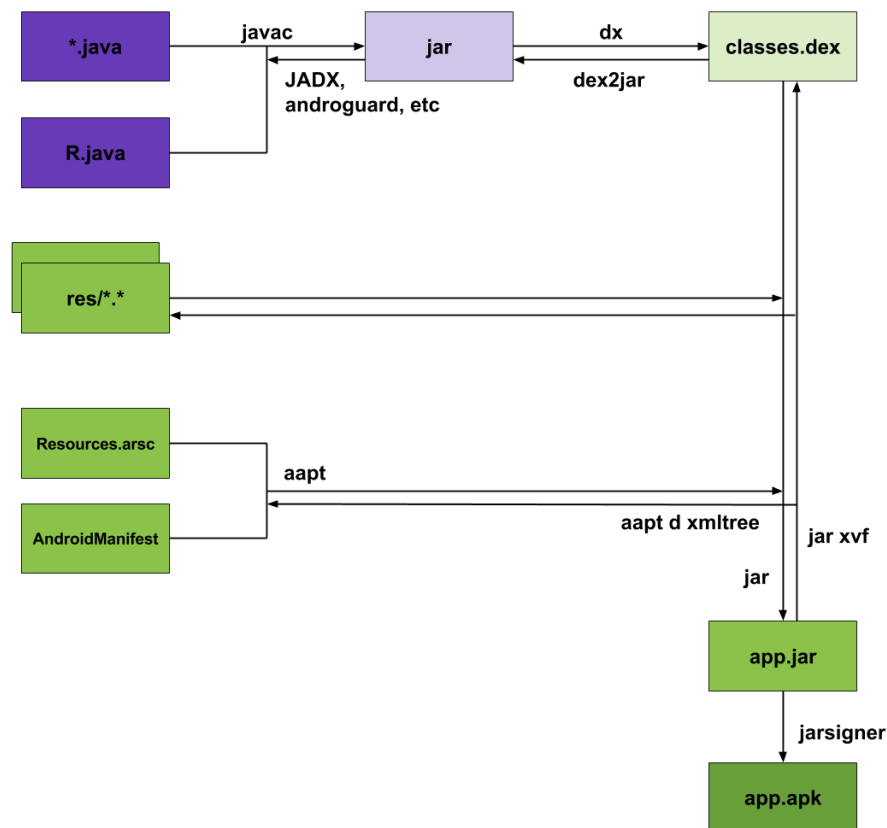


Figure 2.15: Overview of build and reverse engineering of an APK [60]

enabling an analyst to understand the concrete relation between implementation and functionality, optimal output of such a process would be the original source code of the application, not possible in general

Therefore, it is necessary for such a process to provide on the one hand abstract information about structure and inter-dependencies and on the other hand result in very detailed information like bytecode and mnemonics that allow interpretation of implementation

hoffentlich starting points für investigations

java, e.g. read the program code faster

was ist reengineering? wie funktioniert es? was ist das ziel?

reverse engineering process makes use of a whole range of different analysis methodologies and tools.

only consider static analysis tools

IN ORDER TO GET FULL OVERVIEW DEX/SMALI/JAVA -see- WARUM?

WAS MACHEN DIE TOOLS IM ALLGEMEINEN? WOZU BENUTZEN WIR SIE?

- It comes as no surprise that .dex and .class are isomorphic
- DEX debug items map DEX offsets to Java line numbers
- Dex2jar tool can easily “decompile” from .dex back to a .jar
- Extremely useful for reverse engineering – Even more useful for formalice and mischief

- Flow from DEX to JAVA is bidirectional, meaning that an attacker can:
- Decompile your code back to Java
- Remove annoyances like ads, registration
- Uncover sensitive data (app logic, or poorly guarded secrets)
- Replace certain classes with others, potentially malicious ones
- Recompile back to JAR, then DEX
- Put cloned/trojaned version of your app on Play or another market
- ASEC/OBB “solutions” for this fail miserably when target device is rooted.

<https://mobilesecuritywiki.com/>

[https://net.cs.uni-bonn.de/fileadmin/user\\_upload/plohmann/2012-Schulz-Code\\_Protection\\_in\\_Android.pdf](https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf)

main tools

### 2.4.1 Retrieving an APK

5 most apps are installed /data/app, apk can be pulled from there

to get installed applications on phone use android package manager after connecting phone to computer and having ADB tools installed `adb shell pm list packages -f(1)` outputs a list of installed apps in format `<namespace>.<appName>` and an appended number “-1”, each is a folder of installed app containing a base.apk (the application apk) example 2 enthält return von (1)

then find app which should be transferred to computer and use `adb pull /data/app/me.neutze.licensetest1/base.apk` to download app into current folder

in case you have root you can use file manager as solid explorer to access the folder directly and copy apk to user-defined location/send per mail

[64]

In the following there will be an example application to generalise the procedure. The application is called *LicenseTest* and has for our purpose a license verification library included (Amazon, Google or Samsung).

In order to analyse an APK, it has to be pulled from the Android device onto the

computer. First the package name of the app has to be found out. This can be done by using the ADB. Entering example 1 returned example 2

dann auf verschiedenen leveln anschauen mit den folgenden tools

### 2.4.2 dex Analysis

This is my real text! Rest might be copied or not be checked!

nur dex weil die apps im moment so vorliegen

aosp-supplied dexdumo to disassemble dex

[60] always attack dex since the protection mechanism is in there (except JNI?) since apk is zip like decompression tool like 7zip can extract classes.dex from apk file

hexdump to get bytecode

code wie er vorliegt, wenn was geändert wird wird es hier geändert

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP) RESULT OUTPUT dex hex-

```
1 #!/bin/bash
2 #hexdump dex
3 unzip lvttest-version.apk -d /tmp/
4 hexdump -C /tmp/classes.dex >> /dex/version/classes.txt
```

Code Snippet 2.7: Script to extract the .dex bytecode from the APK

dump, start of the dex file, e.g first 8 byte represent the dex header magic dex.035, version 35[17] LINE | bytecode | ASCII representation

```
00000000 64 65 78 0a 30 33 35 00 ae a5 51 7e 06 f7 00 84 |dex.035...Q~....|
00000010 ee 23 5d 3b 4a 61 bb 08 51 a7 c9 02 c1 4e d2 91 |.|#];Ja..Q....N..|
00000020 0c fb 21 00 70 00 00 00 78 56 34 12 00 00 00 00 |...!.p...xV4....|
00000030 00 00 00 00 ac 88 06 00 f4 4e 00 00 70 00 00 00 |.....N..p...|
00000040 ad 09 00 00 40 3c 01 00 0a 0e 00 00 f4 62 01 00 |....@<.....b..|
00000050 3d 27 00 00 6c 0b 02 00 ff 4b 00 00 54 45 03 00 |='.1...K..TE..|
```

Code Snippet 2.8: dex hexdump example

jedes tool:

woher kommt es?

wozu wurde es erfunden?

wer hat es erfunden? quelle

blabla von der seite

wozu benutze ich es?

welches abstrahierungslevel  
beispiel  
additional features?  
WARUM SCHAUEN WIR ES UNS AN?  
wo findet man es?  
welches level?  
vorteil  
blabla aus dem internet

### 2.4.3 Smali Analysis

This is my real text! Rest might be copied or not be checked!

basically jasmin syntax smali, most popular Dalvik bytecode decompilers (used by multiple reverse engineering tools as a base disassembler, amongst which is the also well-known apktool) [56] dex == smali, smali better readable but dex to see how easy change since the translation from java to dex does some optimizations/logik, dex and java do not express the same, but it is how it is in the decompiled code, java is also an abstraction of the actual code, sometimes java also a little confusing since changes happened in dex code and cannot be decompiled to java in a good manner, very messy, it is included for better understanding anyways since humanreadable

stichwort mnemonics, eine seite dex und auf der anderen seite smali, dex bytecode vs smali, Only a few pieces of information are usually not included like the addresses of instructions

unintuitive representation, deswegen smali mit corresponding mnemonics

mnemonics and vice versa is available due to the bijective mapping

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

<https://github.com/JesusFreke/smali>

Smali code is the generated by disassembling Dalvik bytecode using baksmali. The result is a human-readable, assambler-like code

The smali [7] program is an assemblerhas own disassembler called "baksmali" can be used to unpack, modify, and repack Android applications



interesting part for obfuscation and reverse engineering is baksmali. baksmali is similar to dexdump but uses a recursive traversal approach to find instructions  
vorteil? -see- So in this approach the next instruction will be expected at the address where the current instruction can jump to, e.g. for the "goto" instruction. This minimizes some problems connected to the linear sweep approach. baksmali is also used by other reverse engineering tools as a basic disassembler

RESULT OUTPUT: selbe wie dex, jedoch human readable, no big difference, nebeneinanderstellung dex/smali

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)

```
1 #!/bin/bash
2 #baksmali
3 java -jar baksmali.jar -x lvltest-version.apk -o /baksmali/version/
```

Code Snippet 2.9: Script to generate the corresponding smali code for a given APK

```
.method protected onDestroy()V
    .registers 2

    .prologue
    .line 77
    invoke-super {p0}, Landroid/support/v7/app/CompatActivity;->onDestroy()V

    .line 78
    iget-object v0, p0, Lme/neutze/licensetest/MainActivity;->mChecker:Lcom/google/android/vending/licensing/LicenseChecker;

    invoke-virtual {v0}, Lcom/google/android/vending/licensing/LicenseChecker;->onDestroy()V

    .line 79
    return-void
.end method
```

Code Snippet 2.10: smali example

EXAMPLE BESCHREIBEN code snippet 2.10

jedes tool:

woher kommt es?

wozu wurde es erfunden?

wer hat es erfunden? quelle

blabla von der seite  
wozu benutze ich es?  
welches abstrahierungslevel  
beispiel  
additional features?  
WARUM SCHAUEN WIR ES UNS AN?  
wo findet man es?  
welches level?  
vorteil  
blabla aus dem internet

#### 2.4.4 Java Analysis

This is my real text! Rest might be copied or not be checked!

dex different patterns for mobile Usage, java does not really now, thats why different  
java decompiler dex to java .dex and .class are isomorphic dex debug items map dex  
offsets to java line numbers tools like dex2jar can easily decompile from dex to a jar  
extremly useful for reverse engineering, even more so useful for malice

probleme des disassemblers erklären  
interpretations sache  
deswegen zwei compiler  
unterschiedliche interpretation resultiert in flow und auch ob sies können ist unter-  
schiedlich

ectl unterschiede/vor-nachteile  
ggf bezug zu DALVIK/buildprocess (Java wird disassembled und dann assembler)

#### Androguard

An analysis and disassembling tool processing both Dalvik bytecode and optimized  
bytecode

DAD which is also the fastest due to the fact it is a native decompiler, WAS ist dad?  
ERKLÄREN? .dex files was performed with DAD, the default disassembler in the  
Androguard analysis tool, largest successful disassembly ratio

underlying algorithm is recursive traversal

androguard has a large online open-source database with known malware patterns

[56]

<https://github.com/androguard/androguard>  
powerful analysis tool is Androguard  
includes a disassembler and other analysis methods to gather information about a program  
Androguard helps an analyst to get a good overview by providing call graphs and an interactive interface -see- habe nur CLI benutzt  
The integrated disassembler also uses the recursive traversal approach for finding instructions like baksmali, see section 2.2  
one most popular analysis toolkits for Android applications due to its big code base and offered analysis methods -see- quelle, warum

```
1 #!/bin/bash
2 #androguard
3 python androdd.py -i lvltest-version.apk -o /androguard/version/
```

Code Snippet 2.11: Script to decompile to Java using androguard

```
private void doCheck()
{
    this.mCheckLicenseButton.setEnabled(0);
    this.setProgressIndicatorIndeterminateVisibility(1);
    this.mStatusText.setText(2131099673);
    this.mChecker.checkAccess(this.mLicenseCheckerCallback);
    return;
}
```

Code Snippet 2.12: Java code example using androguard

## JADX

```
1 #!/bin/bash
2 #jadx
3 jadx -d /jadx/version/ --deobf --show-bad-code lvltest-version.apk
```

Code Snippet 2.13: Script to decompile to Java using JADX

```
private void doCheck() {  
    this.mCheckLicenseButton.setEnabled(false);  
    setProgressBarIndeterminateVisibility(true);  
    this.mStatusText.setText(C0213R.string.checking_license);  
    this.mChecker.checkAccess(this.mLicenseCheckerCallback);  
}
```

Code Snippet 2.14: Java code example using JADX

### 2.4.5 Detect Code Manipulations

This is my real text! Rest might be copied or not be checked!

vergleich gibts guten einblick was geändert wurde und wie es auf dem gegebenenem lvl funtkioniert

vergleich von original und modifizierten code einer apk auf einer code ebene  
needed to see differences before and after cracking tool

diff is used

```
1 #!/bin/bash  
2 #dex  
3 diff -Naur /dex/original/ /dex/manipulated/ > dex.diff  
4 #smali  
5 diff -Naur /baksmali/original/ /baksmali/manipulated/ > baksmali.diff  
6 #jadx  
7 diff -Naur /jadx/original/ /androguard/jadx/ > jadx.diff  
8 #androguard  
9 diff -Naur /androguard/original/ /androguard/manipulated/ > andro.diff
```

Code Snippet 2.15: Script to compare the original and manipulated APK to see the modifications in the different presentations

erklärung command [87] -N: Treat absent files as empty; Allows the patch create and remove files.

-a: Treat all files as text; Allows the patch update non-text (aka: binary) files.

-u: Set the default 3 lines of unified context; This generates useful time stamps and context.

-r: Recursively compare any subdirectories found; Allows the patch to update subdirec-

tories.  
script erklären

```
@@ Pattern N1 @@  
- 03 01 00 00 0f 00 00 00 1a 00 00 00 0f 00 00 00 |.....|  
+ 03 01 00 00 0f 00 00 00 0f 00 00 00 1a 00 00 00 |.....|
```

Code Snippet 2.16: Diff on Dex level for N1 pattern

RESULT code snippet 3.1  
wo findet man es?  
welches level?  
vorteil  
blabla aus dem internet

## 3 Cracking Android Applications with Lucky Patcher

Cracking apps are a widespread phenomenon on Android these days since root can be acquired easily. There are a number of tools which try to attack and alter Android apps. The resulting piracy thread is discussed a lot in the Android developer community. One of the most popular cracking application is Lucky Patcher, on which this thesis will focus, especially on its license verification bypassing mechanism.

### 3.1 Lucky Patcher

On the official website, Lucky Patcher is described as "[...] a great Android tool to remove ads, modify apps permissions, backup and restore apps, bypass premium applications license verification, and more" [35]. It is written by a developer called ChelpuS and currently on version is 6.0.4 (on 02/17/2016).

Lucky Patcher offers the removing of the licensing in premium apps to crack their DRM, to remove in app ads, change and restrict permissions and activities as well as to create modified after applying one of the feature above on the original APKs [35]. Since copy protection by Google is deprecated, all applications are stored in `/data/app/`. The user as well as Lucky Patcher can access this folder and copy applications from it. *Root* and *busybox*, an application which provides standard UNIX tools for Android[82], is only required when the application should be modified on the device, which is explained later. Lucky Patcher requires no technical knowledge and offers automatic cracking for non professionals. This combination makes it a popular and an effective tool with a high damage potential. [64]

This thesis focuses on how Lucky Patcher is bypassing the license verification mechanism of applications. The goal of circumventing the license check is to make the pirated application work as if had been legally acquired in the store. As described in section 2.3, the license verification is implemented as client-server connection. The app sends information about the user and the application to the server, which checks and verifies the given information, and replies with an approval or rejection. The server communication is secure against man in the middle attacks or spoofing, as messages are private key encrypted [64]. Lucky Patcher is taking a different path by modifying

the application itself. This will be analysed in detail in the following chapters after explaining the general usage.

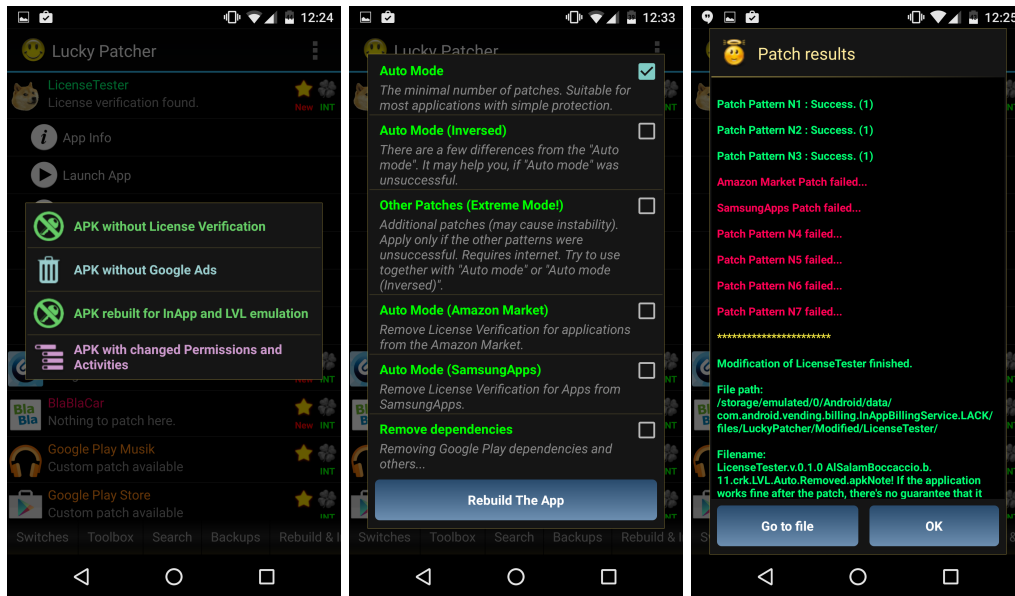


Figure 3.1: Left to right: Features offered LuckyPatcher, modes to crack license verification and the result after patching

Using Lucky Patcher is fairly simple. The application can be downloaded from the official website [35] as an APK and installed on the device. In order to have the features available, the device has to be rooted. On startup of Lucky Patcher, all installed applications are shown in a list. An info text about what patches can be applied is shown below each name. When a the application is selected, a submenu offers various actions, e.g. to get information about the app or run it. Figure 3.1 shows the menu of patches which can be applied to the application. These patches can either be applied in two ways. Choice one is by applying the patch directly on the device and should be taken when *root* is available. This method creates an *.odex* in the Dalvik cache on the phone. Choice two is by creating a modified APK, which does not require *root*. In order to to run the cracked application, the original APK has to be removed and replaced by the modified one. Upon selecting one of the two choices for removing the license verification, different patching modes can be selected as seen in figure 3.1 in the middle. Their description is rather short and does not offer information of how the modes are working. After selecting the mode, the patching is done and the result is shown. As seen in figure 3.1 on the right, different patching patterns are applied

when removing license verification. The different patching patterns are analysed and explained in section 3.4. Lucky Patcher does guarantee that the result is working and all license verification related restrictions are removed.

In this thesis, Lucky Patcher is analysed in two different ways.

- source code reengineering
- analysis of cracked applications and comparison with the original

## 3.2 Code Analysis

The code analysis was done using Lucky Patcher in version 6.0.4 using the two tools (Androguard and JADX) described in subsection 2.4.4. The reversed code was inspected using a text editor like Atom [47].

Before analysing the code, a look is taken at the structure. When loading the folder of the reengineered code into the editor a lot of different code folders can be spotted. These code folders hold the packages of Lucky Patcher itself. libraries used and resources. The libraries can be divided into four categories.

1. Android Support Library v4 with many of it's modules. It is used for downward compatibility of Android related functions of Lucky Patcher itself.
2. Lucky Patcher code. They are located in two places. Utility functions, like "copy file" and "rename", are stored in the package `com.chelpus` utility functions The *application code* itself can be found in the package `com.android.vending.billing.InAppBillingService.LACK`. It contains the activities and functions which are used for cracking applications.
3. The third category are support libraries required by Lucky Patcher to apply it's cracking mechanisms. This includes libraries, e.g. `axml` [36] for serializing the `AndroidManifest.xml` from Android binary into an ASCII formatted, human readable xml and `zip4j` [62], a Java library to handle ZIP files. The fourth and last category is a modified billing and license library. It is applied in combination with a proxy to redirect inapp billing and licensing calls.
4. Already cracked LVL.

Besides the four code folders, the asset folde stores different predefined custom patches which can be applied applications [64].

The analysis is difficult since there is no folder structure sorting the classes in activities or models. The *patchActivity* in the package `com.android.vending.billing.InAppBillingService.LACK` can be identified as launcher method when looking into the `AndroidManifest.xml` file.



The folder embracing the *patchActivity* has a flat hierarchy and is not grouped into sub-folders by activities or models. In the *launcherActivity*, the *listAppsFragment* is created. It implements most of the app logic in over 17.000 lines of code. The additional files are models and classes implementing functions, functions like *CorePatch* for writing the output file. The lack of structure is deliberately chosen to make it hard to get a clear understanding of how exactly Lucky Patcher is patching the attacked application. Another technique chosen by Lucky Patcher is obfuscation of strings and methods which makes it impossible to get their purpose just from the name. At some points it is even impossible for the decompiler tool to disassemble to Java code, forcing it to just return the instructions dump.

Others have already tried to analyse the code of Lucky Patcher. Some of its functions are documented in in Marius-Nicolae Muntean's Master's thesis *Improving License Verification in Android* [64].

### 3.3 Analysis of Patched Applications

Since the analysis of the code is hard and does not uncover its functionality voluntarily, an additional analysis of the reverse engineered code of the outcome of Lucky Patcher is done. The analysis is done to identify the parts of the code manipulated by Lucky Patcher and later on find solutions to protect these weak points. Different applications are used to get a variety of results and see how different implementations are attacked. They are patched and output as modified APK by Lucky Patcher as described in section 3.1. Since the code itself is modified, a static analysis is sufficient. Afterwards they are analysed according to the methodology shown in section 2.4. The analysis is done with modified APKs and not on the device directly since the resulting .odex files are device specific and cannot be used for a general conclusion.

The goal of reverse engineering the code and comparing it to the original application is to see the changes on different levels. This includes the .dex level, on which Lucky Patcher works, the smali level, which makes the .dex code human readable friendly, as well as Java level, on which the functionality change can be identified. On each level the modified and original code will be compared using diff to retrieve the changes in an easy way as well as ignoring the unchanged code.

In order to have a reference application with known source code and an implementation according to the tutorial of LVL [13], a test application is created. This application is called *LicenseTest*. It makes it possible to analyse how Lucky Patcher works on the most basic version. Besides circumventing the Google LVL, Lucky Patcher supports the cracking applications from Amazon and Samsung as well. For Amazon's Kiwi DRM, the same application, with deactivated LVL, is uploaded to Amazon and injected with

their DRM. The analysis for Samsung's Zirconia DRM is only done by using *LicenseTest* with deactivated LVL as well, since the library is implemented the same way into all applications. This can be assumed since the library is included as a .jar and cannot be modified. In addition to the basic application, other applications were analysed as well, to see how Lucky Patcher handles different implementations. The applications, are Runtastic Pro[70], version 6.3, and Teamspeak 3[83], version 3.0.20.2, for the LVL and A Better Camera [5], version 3.35, for the Amazon DRM. These apps were chosen since they were already owned and approved to be included into the thesis by the developers. They include Google's LVL and Amazon DRM.

In addition to the code analysis, the modified application is installed on different devices to evaluate the success of the crack. This is possible since the modified application can be installed on any device. The goal is to identify whether the crack works even though the corresponding store, root or internet connection are not available.

As described before, Lucky Patcher offers different modes to patch applications. Each mode uses a set of patterns which each change a piece of binary. These patterns are shown in figure 3.1 on the right. A pattern is a set of predefined sequences of bytecode in which a certain values are modified. In order to discover applied patterns and to evaluate each mode, each mode is applied on each application.

These are the different modes and what Lucky Patcher describes them as.

- The Auto Mode - "The minimal number of patches. Suitable for most applications with simple protection".
- Auto Mode (Inversed) "There are a few differences from the "Auto mode". It may help you, if "Auto mode" was unsuccessful."
- Other Patches (Extreme Mode!) - "Additional patches (may cause instability). Apply only if the other patterns were unsuccessful. Requires internet. Try to use together with "Auto mode" or "Auto mode (Inversed)"."
- Auto Mode (Amazon Market) - "Removes License Verification for applications from Amazon Market"
- Auto Mode (SamsungApps) - "Removes License Verification for Apps from SamsungApps" (Note: SamsungApps is called GalaxyApps, see subsection 2.3.3)

## 3.4 Patching Patterns

In order to identify the structure of the single patterns, the code of the original code was compared to the cracked output. The changes in the code were inspected on dex, smali

and Java level with the tools explained in Section 2.4. After analysing same patterns in the different modes it was identified that these patterns can be summed up as one. The names of the patterns are taken from the patching result output in figure 3.1 on the right. The number next to the pattern indicates how often it was applied to the application. This number is not always correct. The patterns Nx are used to circumvent the LVL while the Amazon and Samsung patterns are tailored to do the same with the library of the respective store. Additional knowledge on the patterns was gained in the analysis.

Before explaining the patterns in detail, some information has to be provided. When analyzing .dex files, instead of using hexadecimal values like *0x0a*, a simplified presentation as *0a* for improved overview in the diff files is chosen. The opcodes used by the DVM are taken from one of the documentations [67]. When converting .dex files to smali files, the arguments of the opcodes are transferred to variables, e.g. *x* in dex code is *vx* in smali.

Since changing the dex code of an application results in a new checksum, the code has to be resigned as well. This changes can be seen in the diff of the dex files. They are not explicitly mentioned in the analysis since it is no direct change of the attack.

#### Patch Pattern N1

Pattern N1 is present in all patching modes except the solo extreme mode. It targets the *verify()* method of *LicenseValidator* class in the *com/google/android/vending/licensing/* folder. This method is responsible for decrypting and verifying the response from the license server [20].

It can be seen in the dex code in code snippet 3.1 that the blocks *1a* and *0f* are swapped in their order.

```
@@ Pattern N1 @@
- 03 01 00 00 0f 00 00 00 1a 00 00 00 0f 00 00 00 |.....|
+ 03 01 00 00 0f 00 00 00 0f 00 00 00 1a 00 00 00 |.....|
```

Code Snippet 3.1: Diff on Dex level for N1 pattern

When looking at the smali code, the two blocks can be identified as cases of a switch statement. Due to the internal mapping by the language, variables have different names. The swap of switch cases *0x1* and *0x2* can be seen in the diff of code snippet 3.2.

```
@@ Pattern N1 @@
- 0x1 -> :sswitch_e0
```

```
- 0x2 -> :sswitch_d5
+ 0x1 -> :sswitch_d5
+ 0x2 -> :sswitch_e0
```

Code Snippet 3.2: Diff on Smali level for N1 pattern

In the Java code snippet 3.3, not only the syntactic but also semantic changes can be seen. Before the patch, *LICENSED* and *LICENSED\_OLD\_KEY* both were handled as valid since *LICENSED* jumps into the next case. After the patch, *NOT\_LICENSED* starts where *LICENSED\_OLD\_KEY* started before. Now, *LICENSED* and *NOT\_LICENSED* have the same behavior which means even though the response code is *NOT\_LICENSED* it is valid.

```
@@ Pattern N1 @@
case LICENSED:
- case LICENSED_OLD_KEY: handleResponse(); break;
- case NOT_LICENSED: handleError(); break;
+ case NOT_LICENSED: handleResponse(); break;
+ case LICENSED_OLD_KEY: handleError(); break;
```

Code Snippet 3.3: Diff on Java level for N1 pattern (abstracted)

The result is the voiding of the *verify()* switch case since despite the input it always handles it as if the user is verified.

### Patch Pattern N2

As well as pattern N1, N2 is applied in all patching modes, except the solo extreme mode. It is more aggressive since it does not only attack the LVL library, but extends it attacks to other Google Mobile Service (gms) libraries, e.g. *com/google/android/gms/ads/*, as well. The extended analysis of different applications shows the altering of custom libraries as well. An example is AnjLab's inapp billing library [25] in the FKUpdater, located at *com/anjlab/android/iab/v3/Security*, which includes code for the Google inapp billing. This happens since the pattern is applied to other locations as well to counter a moved LVL. Similar to pattern N1, pattern N2 attacks the *LicenseValidator* class's *verify()* method.

The changes in the .dex file can be seen in code snippet 3.4. The sequence *0a 05* is replaced by *12 15*.

```
@@ Pattern N2 @@
- 0c 05 6e 20 9d 4a 53 00 0a 05 39 05 2d 00 1a 05 |..n .JS...9.-...|
+ 0c 05 6e 20 9d 4a 53 00 12 15 39 05 2d 00 1a 05 |..n .JS...9.-...|
```

Code Snippet 3.4: Diff on Dex level for N2 pattern

Smali displays this in a more convenient manner as seen in code snippet 3.5. Instead of moving the result of the proceeding function to *v5* it is always set to *true*.

```
@@ Pattern N2 @@
- move-result v5
+ const/4 v5, 0x1
```

Code Snippet 3.5: Diff on Smali level for n2 pattern

The impact on the Java code (see code snippet 3.6) is more complex than just setting a variable to true. Instead of proceeding only if the signature is verified, the result is ignored and the following code is always executed. The Java code looks different since the decompiler collapses the *if(true)* statement.

```
@@ Pattern N2 @@
- if (sig.verify(Base64.decode(signature))) {...;}
+ sig.verify(Base64.decode(signature)); ...;
```

Code Snippet 3.6: Diff on Java level for N2 pattern (abstracted)

The result is that despite a possibly invalid signature the code of *verify()* is executed anyways.

### Patch Pattern N3

Pattern N3 is different than the other patterns since there are two versions of it. The first version, N3, is used in auto mode while pattern N3i is used in the inversed auto mode. The name N3i is chosen since it is used in the inversed mode. LuckyPatcher does not make a difference between them in the result output. They are combined under the same number since both attack the same logic inside the classes defining the policies. In case of the basic implementation of the LVL, these classes are the *APKExpansionPolicy* and *ServerManagedPolicy* in the *com/google/android/vending/licensing/* folder. Those two classes are examples of policies offered by Google [20]. Pattern N3 attacks their *allowAccess* method.

In case of pattern N3, 01 is replaced with 11, while in case of N3i 11 is replaced by 01 (see code snippet 3.7).

```
@@ Pattern N3 @@
- 12 10 12 01 71 00 a6 89 00 00 0b 02 52 84 c1 1c |....q.....R...|
+ 12 10 12 11 71 00 a6 89 00 00 0b 02 52 84 c1 1c |....q.....R...|

@@ Pattern N3i @@
- 34 00 00 00 12 11 12 00 71 00 70 9d 00 00 0b 02 |4.....q.p.....|
+ 34 00 00 00 12 01 12 00 71 00 70 9d 00 00 0b 02 |4.....q.p.....|
```

Code Snippet 3.7: Diff on Dex level for N3 pattern

When looking at the smali diff in code snippet 3.7, the dex code is translated to the initialization of *v1*. While N3 sets *v1* to 1, N3i sets *v1* to 0.

```
@@ Pattern N3 @@
- const/4 v1, 0x0
+ const/4 v1, 0x1

@@ Pattern N3i @@
- const/4 v1, 0x1
+ const/4 v1, 0x0
```

Code Snippet 3.8: Diff on Smali level for N3 pattern

The resulting code is shown in code snippet 3.9. The pattern attacks the result of the function. While N3 is used to attack code where the default return value is *false*, pattern N3i is used when the default value is *true*.

Both patterns attack the class's *allowAccess()* method which evaluates whether the verification result is according to the policy or not. The return value is only changed in case the test of the input is evaluated accordingly, if this is not the case, it is not changed. The pattern changes the default return value and thus makes the result of the evaluation negligible since the desired result is already set. There are two versions of the pattern to counter inversed logic which can be implemented easily.

### Patch Pattern N4

Pattern N4 was only applied once in the test sample and is part of the auto and auto inverse patching modes. The target of the pattern is the *LicenseChecker* class of the LVL

```
@@ Pattern N3 @@
- result = false;
+ result = true;
return result;

@@ Pattern N3i @@
- result = true;
+ result = false;
return result;
```

Code Snippet 3.9: Diff on Java level for N3 pattern (abstracted)

which is responsible for initializing the license check in its *checkAccess()* method [20]. As seen in code snippet 3.10, it replaces 38 with 33.

```
@@ Pattern N4 @@
- d5 70 00 00 0a 00 38 00 0e 00 1a 00 5a 20 1a 01 |.p....8.....Z ..|
+ d5 70 00 00 0a 00 33 00 0e 00 1a 00 5a 20 1a 01 |.p....3.....Z ..|
```

Code Snippet 3.10: Diff on Dex level for N4 patch

In the smali code snippet 3.11 this change can be identified as replacing *if-eqz* with *if-ne*. Since *if-eqz* evaluates only the one argument *v0* and thus the second argument is 0 in the .dex file, the two argument using *if-ne* interprets this as *v0*.

```
@@ Pattern N4 @@
- if-eqz v0, :cond_15
+ if-ne v0, v0, :cond_15
```

Code Snippet 3.11: Diff on Smali level for N4 patch

The Java code interprets the variable *v0* as boolean. Thus instead of checking whether the result of *mPolicy.allow()* is *false*, it compares the result for inequality with itself, which is always false.

```
@@ Pattern N4 @@
- if( ! mPolicy.allow()) {...}
+ if(mPolicy.allow() != mPolicy.allow()) {...}
```

Code Snippet 3.12: Diff on Java level for N4 patch (abstracted)

The result of patching with pattern N4 is that in the *checkAccess()* method the result, whether the policy allows the proceeding, is never considered.

### Patch Pattern N5

As part of the extreme mode, pattern N5 works on the .dex file similar to pattern N2 on the *LicenseValidator*'s *verify()* method. It does not affect the standard implementation of the LVL.

In the Java diff in code snippet `codeSnippet:n5DiffJava` the result of patching can be seen. Instead of setting the variable to the result of parsing first value of the array *v2\_1*, the value of the array is parsed but the variable is set to 0.

```
@@ Pattern N5 @@
- v0_0.a = Integer.parseInt(v2_1[0]);
+ Integer.parseInt(v2_1[0]);
+ v0_0.a = 0;
```

Code Snippet 3.13: Diff on Java level for N5 patch (abstracted)

The result of this pattern is that inside *verify()* method the check for not equality with 0 is always false. This means the response code is always excepted and thus the exception *Response codes don't match* is never raised.

### Patch Pattern N6

Pattern N6 is part of the extreme mode and, similar to the pattern N1, N2 and N5, it attacks the *verify()* method in the LVL's *LicenseValidator* class.

This pattern changes three values of the .dex file which can be seen in code snippet 3.14. The first changed value is 38 which is replaced by 12. The second value is 06 which is replaced by 00. The third change is the replacing of 4a by 00.

```
@@ Pattern N6 @@
- 38 0a 06 00 32 4a 04 00 33 5a 21 01 1a 00 ab 15 |8...2J..3Z!.....|
+ 12 0a 00 00 32 00 04 00 33 5a 21 01 1a 00 ab 15 |...2...3Z!.....|
```

Code Snippet 3.14: Diff on Dex level for N6 patch

The change of the code structure is more visible in the code snippet 3.15. The first results in the initialization of *p2* with 0. The second change is required since moving a value to a variable only requires two arguments instead of one argument plus the



destination offset. In order to get a valid syntax, the code has to be changed accordingly. The result is interpreted by the decompiler as a *nop* operator and an empty line. The third change is the replacing arguments *p2* and *v4* of the *if-eq* evaluation with *v0* for each.

```
@@ Pattern N6 @@
- if-eqz p2, :cond_e
+ const/4 p2, 0x0
+ nop
+

- if-eq p2, v4, :cond_e
+ if-eq v0, v0, :cond_e
```

Code Snippet 3.15: Diff on Smali level for N6 patch

The analysis on Java level is more complex since removing the *if-eqz* statement leaves the original condition without an entry call. The Java decompiler fails to interpret the missing connection and thus only the *if-eq* is recognizable, see code snippet 3.16. Since this evaluation is always true, the code inside the condition is always executed.

```
@@ Pattern N6 @@
- if (responseCode == 0 || responseCode == NOT_LICENSED || responseCode
    == LICENSED_OLD_KEY) {...} return;
+ if (null == null || broken code return;
```

Code Snippet 3.16: Diff on Java level for N6 patch (abstracted)

This pattern is used to void the license code checks in the *verify()* method.

### Patch Pattern N7

The final pattern for the LVL is pattern N7. It is applied in the extreme mode and as this mode indicates, it takes a harsh approach on applying itself. It does not only patch the *ILicenseResultListener* class's *onTransact()* method, which is the implementation for the callback for interprocess communication and receives the async response from the license server [20]. In addition the pattern is applied to all classes eligible in the *com/android/*. It can be described as the bruteforce version of pattern N2 and thus should only applied when the other modes are not successfully since the outcome might not be stable anymore.

Similar to pattern N2, pattern N7 attacks by initializing the corresponding variable

with *false* instead of moving a result of a method into it.

```
@@ Pattern N7 @@
- x = foo();
+ x = false;
```

Code Snippet 3.17: Diff on Java level for N7 patch (abstracted)

The goal of this attack is to patch the *ILicenseResultListener*, where ever it may be implemented, so *onTransact()* returns true independet of the result of *verifyLicense()*, which is returned in the original implementation.

### Overview for Patching the LVL

The summary of the LVL patterns and their use in the patching modes can be seen in table 3.1. The auto mode and inversed auto mode are applying patches at the important parts of the LVL. They are very efficient as long as the library is not modified by the developer.

In contrast to the determined patching of the automatic modes, the extreme mode tries to apply patterns to different places as well as to more complex methods. This might cause instability, as seen in pattern N6, since it alters the syntax of the dex file.

Modus	Patterns							
	N1	N2	N3	N3i	N4	N5	N6	N7
Auto	X	X	X		X			
Auto (Inversed)	X	X		X	X			
Extreme						X	X	X
Auto+Extreme	X	X	X		X	X	X	X
Auto (Inversed)+Extreme	X	X		X	X	X	X	X

Table 3.1: Patching patterns applied by each mode

### Amazon Market Patch

Amazon does not have different pattern. Since the Kiwi library is injected by Amazon and cannot be customized by the developer, only one pattern is necessary. The pattern is applied twice. The first class tageted is *com/amazon/android/licensing/b.java* while the second class is *com/amazon/android/o/d.java*. Since the injected library is obfuscated, no meaningful names for the methods can be presented.

The Amazon pattern works similar to the LVL's pattern N4 and replaces (38) with (33).

```
@@ Pattern A @@
- 0a 00 38 00 0a 00 62 00 56 20 1a 01 4e 49 6e 20 |..8...b.V ..NIn |
+ 0a 00 33 00 0a 00 62 00 56 20 1a 01 4e 49 6e 20 |..3...b.V ..NIn |
```

Code Snippet 3.18: Diff on Dex level for Amazon patch

The pattern replaces *if-eqz* with *if-ne* as seen in code snippet 3.11. Since *if-eqz* evaluates only the one argument *v0* and thus the second argument is *0* in the .dex file, *if-ne*, which requires two arguments, interprets this as *v0*.

```
@@ Pattern A @@
- if-eqz v0, :cond_1f
+ if-ne v0, v0, :cond_1f
```

Code Snippet 3.19: Diff on Smali level for Amazon patch

The Java code makes it easier to identify the functional changes of the attack. Since the if evaluation is always wrong inside the *b.java* class, the case when the response code is not *LICENSED* is never called. The same changes are applied to the *d.java* class but since the variables are obfuscated it is hard to describe the exact behavior. After analysing the dependencies, it can be said that the function checks whether the given string is not null and then returns true. After patching, the result is always true since the null check is always false as in *b.java*.

```
@@ Pattern A @@
- if( ! v0.equals("LICENSED")) {...}
+ if(v0.equals("LICENSED" != v0.equals("LICENSED")) {...}
```

Code Snippet 3.20: Diff on Java level for Amazon patch (abstracted)

The analysis of the Amazon patch indicates that there are less patterns needed since there is no modified code to expect. Patches are applied to manipulate the checks in case the response code is null or different than *LICENSED*. The result is forced to be always true and thus the license verification always passes.

### Samsung Market Patch

Similar to the Amazon's Kiwi library, Samsung's Zirconia library cannot be modified as well. Thus cracking the library only requires one patch as well. The patch is applied on the *LicenseRetriever* and *Zirconia* class in the *com/samsung/zirconia* package. Unlike

Amazon's Kiwi DRM, Zirconia is not obfuscated and thus better to understand. The Samsung patch uses two patterns, called S1 and S2 in order to distinguish between them. While S1 is applied on both classes, S2 is applied twice but only on the *Zirconia* class.

While Pattern S1 replaces *d6* with *00*, pattern S2 uses *12* instead of *0a*.

```
@@ Pattern S1 @@
- 08 00 0c 08 6e 10 66 4a 08 00 0a 06 32 d6 0a 00 |....n.fJ....2...|
+ 08 00 0c 08 6e 10 66 4a 08 00 0a 06 32 00 0a 00 |....n.fJ....2...|

@@ Pattern S2 @@
- 10 02 0a 00 0f 00 00 00 03 00 01 00 02 00 00 00 |.....|
+ 10 02 12 10 0f 00 00 00 03 00 01 00 02 00 00 00 |.....|
```

Code Snippet 3.21: Diff on Dex level for Samsung patch

The result of pattern S1 is that the *if-eq* statement is always true since the code now evaluates the equality of *v0* compared with itself, which is always *true*. Pattern S2 has the effect that *v0* does not contain the result of a method but always *true*.

```
@@ Pattern S1 @@
- if-eq v6, v13, :cond_52
+ if-eq v0, v0, :cond_52

@@ Pattern S2 @@
- move-result v0
+ const/4 v0, 0x1
```

Code Snippet 3.22: Diff on Smali level for Samsung patch

The abstract presentation of the result in Java code in code snippet 3.23 shows the resulting behavior.

Instead of checking the response code for validity, *LicenseRetriever*'s *receiveResponse()* method always skips the check, when pattern S1 is applied, and executes as if it was valid. In the method *checkerThreadWorker()* of the *Zirconia* class, pattern S1 voids the check of the response code and always continues as if the response code was valid.

Pattern S2 works on the methods *checkLicenseFile()* and *checkLicenseFilePhase2()* of the *Zirconia* class. Instead of returning the result of the license check, the methods return always *true*.

```

@@ Pattern S1 @@
- if (v6 == foo()) {...}
+ foo()
+ if (v0 == v0) {...}

@@ Pattern S2 @@
- return foo();
+ return true;

```

Code Snippet 3.23: Diff on Java level for Samsung patch (abstracted)

The result of applying the patch is that not only the license file checks are voided and return true as default, but also reponse codes other than *LICENSED* are accepted since they are neither checked for validity nor to the stored one, which should be valid since it was stored.

### 3.5 Conclusion and Learnings

Results for applications, was ist wenn auf handy ohne store/mit store root/ohne root etc - ringschluss blackmarket da man installieren kann

Modus	Application		
	LicenseTester	Runtastic Pro	Teamspeak 3
Purchased	yes	yes	yes
Pirated	no	no	no
Auto	yes	yes	no
Auto (Inversed)	no	yes	no
Extreme	no	yes	no
Auto+Extreme	yes	yes	no
Auto (Inversed)+Extreme	no	yes	no

Table 3.2: Functionality for the test apps before and after patching

lvl cracked app was always workign (test)

Amazon cracked app was always working (no client needed) samsung app also working

applies it directly to the librabry since first patching point could be the initial call, in case modified lvl patching initial call would be not enough since the on success block could contain important code (like ui creation) then it would be useless, target on specific points where decisions are made to alter as few code as possible

since automated customizations have to be implemented to trick it make false checks to detect tampering -see- user patch

amazon/samsung not much to do since from company, beyond control of developer since injection after developer and a library provided by samsung which is only called, that is why the following not simple methods target lvl

known bytecode patterns, replace with custom, makes mechanism useless

following present ways of protecting against patching attempts, especially predefined recipes circumventing the LVL high motivation, the patterns/patching modes cover many apps, more than custom

should not use one but many methods solution for current version of lucky patcher, future might be different, arms race scenario [64]

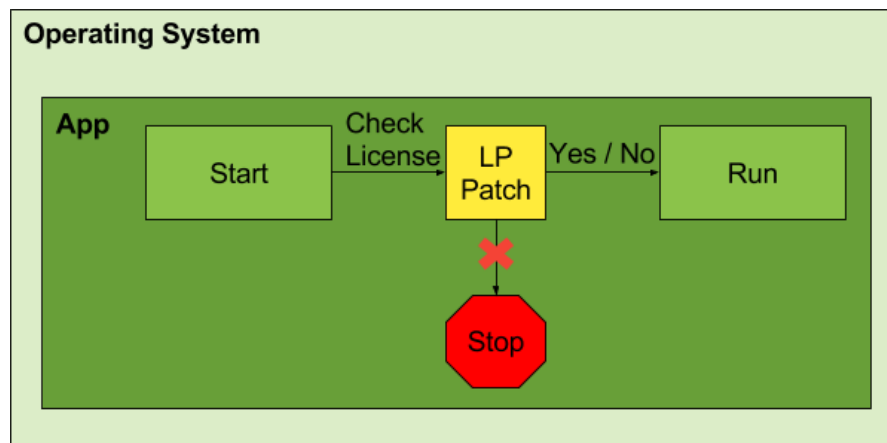


Figure 3.2: Abstraction of the current attack on the license verification mechanism

Aamzon + Samsung Wenn man sich slide.me anschaut, ist dies eine simple Variante von Zirconia und ist theoretisch noch einfacher zu cracken. Anstatt auf einen Callback zu setzen welchen man in der jar modifizieren müsste, kann man bei slide.me einfach den Code Block `if(someRights != null) // you have granted rights. else // You don't have any rights for the feature in cause. Try // some features.` (Currently not supporting multiple 'features') modifizieren und immer in den "you have granted rights" Block springen. Dies ist besonders einfach, da durch die statische jar auch kein modifiziertes Verhalten zu erwarten ist.

nur ja/nein test bzw ergebnis zuweisung und drauf folgender test kann IMMER geskippt werden, vergleich figure 2.13

unary outcome beschreiben

fügt keinen code hinzu sondern ersetzt commands => checksum/signature

## 4 Countermeasures for Developers

Now that the methodology of Lucky Patcher is analyzed, improvements to prevent it from circumventing the license check mechanism can be proposed. The first chapter will cover methods to improve the current implementation of the LVL, additional integrity tests to detect tampering and options to make the code more resistant against attacks.

The second chapter proposes ways to make the unary outcome when the license is verified more complex by introducing a content server and encryption.

The third chapter takes a look at ART and Trusted Execution Environment (TEE) as environmental improvements to tackle this issue.

### 4.1 Extend Current Library

The first action is to fortify the spots identified in section 3.4 as being attacked by Lucky Patcher. The goal is to prevent the automatic application of patching patterns and stop execution in case tampering was detected. Since additional checks can be voided by analysing the code manually and adding them to the patching procedure, obfuscation is introduced as a tool to make reverse engineering more complex.

#### 4.1.1 Modifications on the Google LVL

Attacking with Lucky Patcher is often successful since many developers do not customize the LVL. One reason for this is developers using the library only against casual piracy. It prevents users, who try to copy APKs directly from one device to another, from using the application. Another reason is that they do not know where exactly to fortify the library and thus they do not want to spend additional work. [53] [64] This thesis presents two approaches for making attacks on the library less successful.

The first approach is to actively go against Lucky Patcher's patterns by modifying the identified parts of code.

The second approach is to implement the LVL with native code which cannot be targeted by patterns.

### Modify the Library

Lucky Patcher's patching is reliant on applying its patterns. Patching application code is the most effective and most common attack to modify the license logic. Going actively against these patterns should always be the first step to challenge Lucky Patcher. This can only be applied to the LVL since it is the only library where the source code can be accessed by the developer. For Amazon and Samsung, this propose can be useful as well but it is more difficult to implement since these two libraries have only one entry and exit point in the applicaiton.

Modifying and improving the LVL does not only protect from patterns. Increasing complexity of the application's bytecode makes it unqiue and harder to reengineer. [53] There are three areas the developer should focus on when modifying the LVL [53].

1. core licensing library logic
2. entry andexit points of the licensing library
3. invocation and handling of the response

The core logic two main classes are the *LicenseChecker* and the *LicenseValidator*. As seen in section 3.4, these two classes are the primary target of Lucky Patcher and thus should be altered as hard as possible while retaining the original function. The isomorphic code changes can include:

- replace the switch statement with an if statement and add additional code between the if statements (see pattern N1)
- use functions to create new values for constants used and check for these values in the further proceeding (see pattern N3)
- remove unused code, e.g. implement the *LicenseValidator* online (see patterns N2, N4, N5, N6)
- move the LVL package into the application (see patterns N2,N7)
- use additional threads to handle different steps in the license verification process
- implement functions inline where possible (see patterns N2, N5, N7)
- make actions in the decompiled code difficult to trace by removing functions or moving routines to unrelated code, counter intuitive from traditional software engineering
- implement radical response handling, e.g. kill the application as soon as a invalid response can be detected, results in bad user experience



These are only examples and creativity is welcome since the resulting implementation should be unique. [53]

The entry and exit points can be attacked by creating a counterfeit version of the LVL that implements same interface. An unquite implementation provides resilience against this attack. It can be achieved by adding additional arguments to the *LicenseChecker* constructor as well as the *allow()* and the *dontAllow()* methods. [53]

Attackers do not only target the LVL but the handling of the result in the application as well. This can be prevented by handling the mechanism in a separate activity. In the original activity *finish()* will be called and the attacker will be stuck in the new activity. This prevents from scenarios where the attacker voids methods which would prevent further proceeding. In addition the license verification can be postponed to a later point in time since attackers are expecting it on the the applications launch. [53]

These modifications are easy to apply and make the implementation unique. The unlimited ways to implement make it hard to attack automatically. This does not ensure total protection against Lucky Patcher since every application is patchable in some way. A determined attacker, willing to invest a lot of time and work in disassembling and reassembling, can eventually hack the implementation. The aim of the developer is to make the work of the attacker as hard as possible to the point where the profit is not worth the time. [53]

#### 4.1.2 Native Implementation of LVL

Lucky Patcher's automatic patching modes, as described in chapter 3, target the application's .dex file. Since Android supports the Native Development Kit (NDK), it is possible to implement parts of the application using native code.

Usually the NDK targets CPU intensive tasks, such as game engines and signal processing, but it can be used for any other purposes as well. Google suggests using it only if necessary since it increases the complexity of an application [14]. In case of the LVL this is a desired side effect. Native code, in opposite of byte-code, does not contain a lot of meta-data, such as local variable types, class structure. It information discarded when the code is compiled and thus makes it harder to understand the code.

There are two scenarios for creating a native implementation of the LVL.

- The developers creates his own version of native implementation
- Google provides a native implementation of the LVL

When the developer creates his own implementation of the LVL it is unique. In order to achieve this, the developer needs the required knowledge and skill as well as time to implement it. In case the implementation is done in a right way, it offers uniqueness

and safety. The attackers have to invest time to analyse the native code of the license verification and the implementation into the application itself. Only after reverse engineering the native code, they can start with searching for a way to break the license verification mechanism, repack the native code and make it available as a custom patch for Lucky Patcher. This scares off attackers since the circumventing of the license verification requires a lot of knowledge and time. As long as the attacker is evaluating the workload with the gain of cracking the application, it is considered a safe method, implied the developer has enough available resources. [64]

Instead of providing Java code, Google could provide a native version of the LVL. In the beginning, it would be harder to find vulnerabilities than it was with the Java version. It would take some time for the attackers to crack the library but it would be justified for the attackers since the library would be implemented into all Play Store applications. Now this license verification would face the same problems as Amazon's or Samsung's libraries since one custom patch applied by Lucky Patcher would be able to crack all applications. For this reason, the implementation should include two essential features.

- heavy obfuscation should be applied since users do not need to understand the library
- encryption, dynamic code generation and automatic customization everytime it is loaded, since having only one version is an easy target

In addition to making the license check native, parts of the application should be moved into the native code. This protects against attacks where the call of the license verification library is skipped.

In general, the proposal is simple, but the implementation is much harder. Until now, no big company has come up with such approach. This indicates that still a lot of research and work has to be done to implement this solution. At least, first steps in this direction have been made by addressing dex and its vulnerabilities. [64]

As long as the license verification library is implemented in native code, the automatic patching modes of Lucky Patcher do not work. Attackers have to analyse the native code and create a custom patch for the application. These custom patches can already be found for some applications in form of modified `.so` files.

### 4.1.3 Tampering Protection

The correct execution can only be guaranteed as long as the integrity of the environment is ensured. When circumventing the LVL the code has to be modified. Unless done precisely, this can be detected, e.g. Lucky Patcher does not have the developers signature and thus uses a different one. Other breaches of integrity are debuggability

and *root*, cracking applications and unauthorized installation. In order to be able to detect this and prevent the execution of the application a priori, additional checks are implemented as seen in figure 4.1. Since all tampering countermeasures have the same

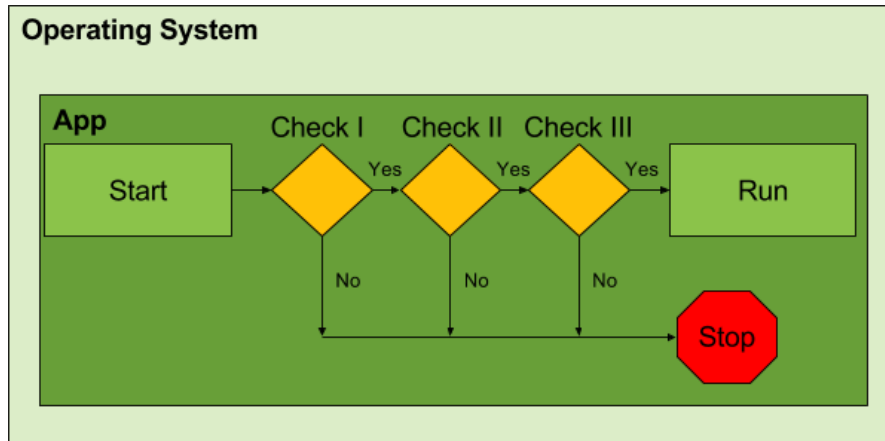


Figure 4.1: Introduction of additional tests to check environment and integrity of the application

pattern in operation, they can be circumvented easily. The goal of these checks is to increase the workload of an attacker since the code has to be analysed in order to find, understand and patch them. In order to make this task even more time consuming, these checks can be obfuscated and spread inside the application. Even the possibility of implementing them in native code can be considered.

This does not prevent Lucky Patcher itself from working, but it offers an additional layer of security which has to be voided.

### Debuggability

Enabling debugging allows the developer to use additional features for analysing the app at runtime, like printing out logs [18]. This features are used to gain information about the flow of the app and to reengineer functionality. From the results of this analysis weak points in the application can be identified and custom patches for cracking tools can be derived from them. This is the reason for disabling the debug flag in release builds on the application stores, but since the flag for debugging is included inside the .dex file, it can be enabled by attacks In order to prevent attackers taking advantage from this possibility, the developer should check whether this flag is activated und thus the application is tampered.

Code snippet 4.1 is an example for an implementation of this check. The debug flag

```
14 public static boolean isDebuggable(Context context) {  
15     boolean debuggable = (0 != (context.getApplicationInfo().flags & ApplicationInfo.  
        FLAG_DEBUGGABLE));  
16  
17     if (debuggable) {  
18         android.os.Process.killProcess(android.os.Process.myPid());  
19     }  
20  
21     return debuggable;  
22 }
```

Code Snippet 4.1: Example code for checking for debuggability

can be acquired from the application information as seen in line 15. In case the debug is set, and thus the application is tampered, the process is killed in line 18.

### Root

Root is the foundation to cracking applications and thus when the device is rooted it possible to alter the application. The developer can check whether root is available on the device and eventually exclude users accordingly. When using this feature, the developer has to communicate the users the reasons for this strict policy since there are a lot of users who use root for other reasons than cracking applications. Google has introduced a similar APK, called SafetyNet [57], which is said to be used in security critical applications like Android Pay [16] [69].

Since root is achieved by moving the *su* file to the file system, the application can search in the common locations for it. In case the search is successful, the execution of the application can be terminated.

### Lucky Patcher

Having Lucky Patcher installed is an indicator that the user is attacking license verifications libraries. The check is not as strict as the one for root and excluding the user from executing is builds upon a foundation. It can be extended to detect additional unwanted applications by adding their package name to the check [90]. Some custom Android versions already include a library called *AntiPiracySupport* [39] which is used to remove and blacklist piracy applications.

As shown in code snippet 4.3, the check tries to acquire whether the Lucky Patcher package is installed. In case information is available and thus the application is installed, the check stops the application.

```
16 public static boolean findBinary(Context context, final String binaryName) {
17     boolean result = false;
18     String[] places = {
19         "/sbin/",
20         "/system/bin/",
21         "/system/xbin/",
22         "/data/local/xbin/",
23         "/data/local/bin/",
24         "/system/sd/xbin/",
25         "/system/bin/failsafe/",
26         "/data/local/"
27     };
28
29     for (final String where : places) {
30         if (new File(where + binaryName).exists()) {
31             result = true;
32             android.os.Process.killProcess(android.os.Process.myPid());
33         }
34     }
35
36     return result;
37 }
```

Code Snippet 4.2: Example code for checking for root

### Sideload

APKs can be cracked on a different device, transferred and installed on the device, so checking for root and Lucky Patcher are not enough. Usually, applications, which include a license verification library, are purchased from the corresponding store. Installing them from other sources is a sign for piracy. For this reason, developers should enforce installation from trusted sources to ensure that the application is purchased as well.

The code snippet 4.4 shows the implementation for the stores in scope for the thesis. Additional stores can and should be added in case the developer decides to offer the application in another store.

This feature should be implemented with caution since Google notes that this method relies on the *getInstallerPackageName* which is neither documented nor supported and only works by accident [53].

```
9  public static boolean checkInstall(final Context context) {
10     boolean result = false;
11     String luckypatcher =
12         // Lucky patcher 6.0.4
13         "com.android.vending.billing.InAppBillingService.LUCK",
14     };
15
16     try {
17         info = context.getPackageManager().getPackageInfo(luckypatcher, 0);
18
19         if (info != null) {
20             android.os.Process.killProcess(android.os.Process.myPid());
21             result = true;
22         }
23
24     } catch (final PackageManager.NameNotFoundException ignored) {
25     }
26
27     if (result) {
28         android.os.Process.killProcess(android.os.Process.myPid());
29     }
30
31     return result;
32 }
33 }
```

Code Snippet 4.3: Example code for checking whether Lucky Patcher is installed on the device

### Signature

Application code is signed to connect a developer to an application and enable him to provide updates for the application. Since the checksum and signature have to be rewritten when cracking the application, this can be used to detect attacks. This is similar to Google Maps inside an application. The Maps library sends the signature and the API key to the server which, similar to the license verification, decides whether the application is allowed to show the map.

The code for the signature check can be seen in code snippet 4.5. In order to check the application's signature, the original signature has to be provided (see line 53). The application's signature fetched from the package information (line 57ff). In case the signature cannot be retrieved or the signature do not match, the check terminates the application.

```
15 public class Sideload {
16     private static final String PLAYSTORE_ID = "com.android.vending";
17     private static final String AMAZON_ID = "com.amazon.venezia";
18     private static final String SAMSUNG_ID = "com.sec.android.app.samsungapps";
19
20     public static boolean verifyInstaller(final Context context) {
21         boolean result = false;
22         final String installer = context.getPackageManager().getInstallerPackageName(context.
23             getPackageName());
24
25         if (installer != null) {
26             if (installer.startsWith(PLAYSTORE_ID)) {
27                 result = true;
28             }
29             if (installer.startsWith(AMAZON_ID)) {
30                 result = true;
31             }
32             if (installer.startsWith(SAMSUNG_ID)) {
33                 result = true;
34             }
35         }
36         if(!result){
37             android.os.Process.killProcess(android.os.Process.myPid());
38         }
39         return result;
40     }
```

Code Snippet 4.4: Example code for checking the origin of the installation

### Flow Control

Another approach to ensure that the application is not tampered is to define a flow inside the application. This is not directly a check but enforces the correct sequential execution of code. The flow has a known sequence and should include which is likely to be tampered. Inside this code, important parts of the application can be interwoven, e.g. the interface initialization. As soon as a cracking tool alters the license library and thus the predefined flow, some code blocks are skipped and thus the essential code inside them is not executed.

A similar technique would be to have two license verifications inside the application. One is known to fail while the other one is the actual check. Since Lucky Patcher does not know the difference, its automatic mode patches both. The developer can take advantage of this by inverting the callback for the result of the known failing verification.

```
51 public static boolean checkAppSignature(final Context context) {
52     //Signature used to sign the application
53     static final String mySignature = "...";
54     boolean result = false;
55
56     try {
57         final PackageInfo packageInfo = context.getPackageManager().getPackageInfo(context.
58             getPackageName(), PackageManager.GET_SIGNATURES);
59
60         for (final Signature signature : packageInfo.signatures) {
61             final String currentSignature = signature.toCharsString();
62             if (mySignature.equals(currentSignature)) {
63                 result = true;
64             }
65         } catch (final Exception e) {
66             android.os.Process.killProcess(android.os.Process.myPid());
67         }
68
69         if (!result) {
70             android.os.Process.killProcess(android.os.Process.myPid());
71         }
72
73         return result;
74     }
```

Code Snippet 4.5: Example code for checking the signature of the application

#### 4.1.4 Obfuscation

now that the lvl is modified and the environment is enforced, the next goal is to prevent pirates from even starting to analyze the applcation, automatischer crack nicht mehr funktioniert, muster, andere mechanismen erkennt er nicht ref kapitel 1 macht gg lucky sicher, aber crakcbar wenn manuell, was kann man dagegen tun

does not help when standard version is implemented, that is why this is working best with customized implementation of LVL

man versucht sich vor neuen angriffen zu schützen indem man reengineering zeitintensiver macht developer can make hackers task immensely more difficult to the point where it is not worth the time[53] reengineering cannot be vermiede[64] best is to apply technquies to make it as hard a possible[64] obfuscate to make it difficult tor everse engineer[53] it is not possible to 100 percent evade reengineering, but adding different methods to hide from plain sight of reengineering tools[64]

Reverse engineering and code protection are processes which are opposing each



other, neither classified as good nor bad[56] "good" developer: malware detection and IP protection[56] "bad" developer: analysis for attack and analysis resistance[56] [56]

Application developers are interested in protecting their applications. Protection in this case means that it should be hard to understand what an application is doing and how its functionalities are implemented. if they do not see what the app is doing, they cannot fix it

Reverse engineering of Android applications is much easier than on other architectures -see- high level but simple bytecode language Obfuscation techniques protect intellectual property of software/license verification possible code obfuscation methods on the Android platform focus on obfuscating Dalvik bytecode -see- limitations of current reverse engineering tools Without proper naming of classes and methods it is much harder to reverse engineer an application, because in most cases the identifier enables an analyst to directly guess the purpose of the particular part. The program code itself will not be changed heavily, so the obfuscation by this tool is very limited.

SOLUTION obfuscators are applied when compiling definition obfuscation, was macht es, wie funktioniert es, wer hat es erfunden, wie wendet man es an first line of defense, can be applied without much extra workload[53] will not protect against autoamted attack, does not alter flow of program[53] makes more difficult for attackers to write initial attack[53] removing symbols that would quickly reveal original structure[53]

(a) at source code and (b) bytecode level, Most existing open-source and commercial tools work on source code level[56] Java code is architecture-independent giving freedom to design generic code transformations. Lowering the obfuscation level to bytecode requires the algorithms applied to be tuned accordingly to the underlying architecture[56]

number of commercial and open-source obfuscators available for Java that will work with Android[53] a few dex obfuscators exist, with different approaches[60] proguard or sdex, rename methods, field and calss names – break down string operations so as to chop hard coded strings or encrypt – can use dynamic class loading (dexloader classes to impede static analysis)[60] can add dead code and dummy lopps (minor impact of performance)[60] can also use goto into other onstructions[60]

be aware that certain methods cannot be obfuscated, even with advanced tools[53] reliance on android framework apis (remain unobfuscated) e.g. on create cannot be renamed since it needs to remain callable by the android system[53] avoid putting license check code in these methods since attackers will be looking for the lvl there[53]

does not protect directly versus luckypatcher but in case of an custom implementation it makes the process of analyzing the app more time consuming in theory a good addition to the security of the application, but against luckypatcher directly since it works on java level in order to disguise the way the code works it enforces increased

initial effort an attacker has to spend in order to understand the code and thus reduces the likelihood of attackers to being motivated to crack the application

### Proguard

This is my real text! Rest might be copied or not be checked!

A Java source code obfuscator. ProGuard performs variable identifiers name scrambling for packages, classes, methods and fields. It shrinks the code size by automatically removing unused classes, detects and highlights dead code, but leaves the developer to remove it manually [56]

open source tool shrinks, optimizes and obfuscates java .class files result - smaller apk files (use rprofits download and less space) - obfuscated code, especially layout obfuscation, harder to reverse engineer - small performance increase due to optimizations integrated into android build system, thus easy use default turned off minifyEnabled true proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

additional step in build process, right after java compiler compiled to class files, Proguard performs transformation on files removes unused classes, fields, methods and attributes which got past javac optimization step methods are inlined, unused parameters removed, classes and methods made private/static/final as possible obfuscation step name and identifiers mangled, data obfuscation is performed, packages flattened, methods renamed to same name and overloading differentiates them

after proguard is finished dx converts to classes.dex

[64]

identifier mangling, ProGuard uses a similar approach. It uses minimal lexical-sorted strings like a, b, c, ..., aa, ab, original identifiers give information about interesting parts of a program, Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed -see- neutralizing these information in order to prevent this reduction, remove any meta information about the behavior, meaningless string representation holdin respect to consistence means identifiers for the same object must be replaced by the same string, advantage of minimizing the memory usage, e development process in step "a" or step "b"

string obfuscation, string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog, information is context, other is information itself, e.g. key, url, injective function and deobfuscation stub which constructs original at runtime so no behaviour is changed, does not make understanding harder since only stub is added but reduces usable meta information

[73]

ProGuard is an open source tool which is also integrated in the Android SDK, free ProGuard is basically a Java obfuscator but can also be used for Android applications

because they are usually written in Java // feature set includes identifier obfuscation for packages, classes, methods, and fields was kann er noch? -see- Besides these protection mechanisms it can also identify and highlight dead code and removed in a second, manual step Unused classes removed automatically by ProGuard. easy integration[4]

optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code merges identical code blocks performs optimizations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [46] [86]

### **Dexguard**

This is my real text! Rest might be copied or not be checked!

commercial Android obfuscator for bytecode and source code various techniques including strings encryption, encrypting app resources, tamper detection, removing logging code [56]

son of proguard, does everything that proguard does its a optimizer and shrinker, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), application tamper protection, removes debug information may increase dex size, memory size; decrease speed [86]

obfuscation methods are a superset of ProGuards more powerful but also does not protect from disassembling the code

protects apps from reverse engineering and ahckign attacks makes apps smaller and faster specialized fr android protects code: obfuscation, hides sensitive strings,keys and entire algorithms [51]

### **Allatori**

This is my real text! Rest might be copied or not be checked!

commercial Android obfuscator name obfuscation, control flow flattening/obfuscation, debug info obfuscation, string encryption the result is a decreases dex size, memory, increases speed, removed debug code+ like Proguard+string encryption [86]

commercial product from Smardec addition to what Proguard does it offeres methods to provide program code, loops are modified so reengineering tools do not recognize it as a loop adds complexity to algorithms and increases their size string obuscation

[2] [3]

## 4.2 Complex Verification

A part of the license verification's insecurity is the fact that not only the possible outcome is binary, either the license is verified or it is not, but it is also possible to predict the outcome since in the end, only one result is accepted. This mechanism can be easily attacked. The attacker can modify that even the wrong result is always accepted.

In order to fight this major flaw, more complex mechanisms have to replace the original license verification. In the following, different approaches are proposed. The first proposal is the introduction of content server which allows to move part of the functionality, like the license verification, from the application to a server. The server cannot be attacked by the cracking application and thus is safe.

The second proposal is to introduce encryption. Encryption enables unpredictable outcomes instead of binary ones.

### 4.2.1 Content Server

An approach to fight the shortcomings of the license verification libraries is moving it to a server. The introduction service managed accounts, users have to login on a server in order to verify their license. Instead of returning the result of the verification, the server delivers the content of the application. Since the license verification is no longer inside the application .dex file, Lucky Patcher is not able to manipulate it anymore.

The implementation can be described with the application Spotify [76] as a reference.

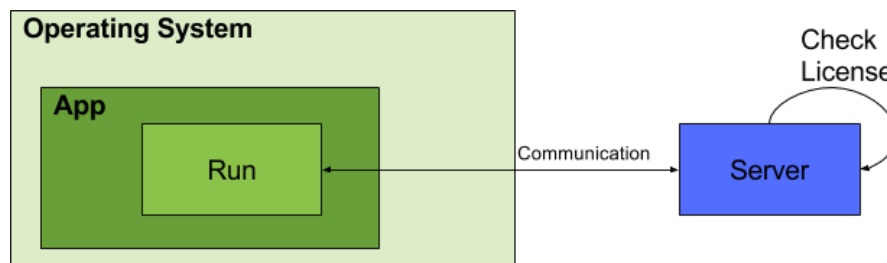


Figure 4.2: Abstraction of an application and a content server

Instead of using local license verification, the user has to enter his credentials, which are sent to the server. In case the credentials are valid, the user is logged into the application. The content, the music in this case, is no longer on the phone itself, but streamed from the server. The attacker still can circumvent the user verification inside the application, but they are facing a second layer of security afterwards. Since the content is on the server and the user has to be authorized on it, no content is available

inside the application. Thus attacks on the applications itself do not work anymore that easily.

Having a content server is good solution against piracy but it has downsides as well. The first problem is that this model cannot be applied to all applications universally. It means that it must be possible to extract parts of the application's logic and implement them on a server. This is not possible for all applications. The second problem are the additional resources needed. When outsourcing parts of the application on a server, not only knowledge and money is needed for the server, but an additional application for the server has to be created as well. Not every developer can handle this additional workload. The third problem is the resulting always online necessity which limits the freedom of users and creates additional unnecessary traffic. This is not accepted by all users.

Nevertheless, if this implementation can be realized, it is safe from Lucky Patcher autopatching as well as custom patches. In addition, this mechanism protects the developers IP when the core algorithm is moved to the server. This prevents attackers not from only using the application for free, but also from reconstructing the the core functionality and implementing it somewhere else, thus the application gives less incentives for attackers.

### 4.2.2 Encryption

Another approach to is the introduction of more complex license verification mechanism by using encryption. The advantage of encryption is the unpredictable outcome of an input which is not spoofable anymore compared to the binary check. Before taking advantage of encryption, it has to be decided what content should be encrypted and where the key should be stored.

#### Encryption

Encryption can be applied on different levels inside the application. It has to be decided to which extend it should be applied. The thesis introduces three different approaches on encryption.

#### Resource Decryption

The first approach is to apply encryption on the application's static resources. This includes the application's hard coded strings as well as image assets. Whenever a resource is used, it has to be decrypted first. The increase in security comes at the cost of decreased performance. As long as application critical strings, like server addresses are encrypted, the application is unable to work. In case no critical strings are present,

the application will work as usual, but the user experience will be not sufficient because all strings are still encrypted and thus have no meaning. Figure 4.3 shows the abstract implementation of resource decryption.

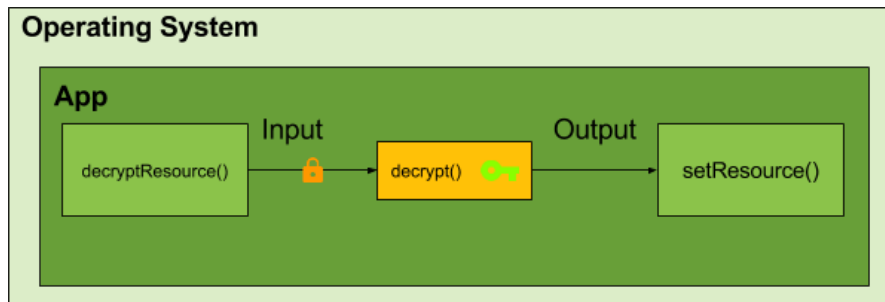


Figure 4.3: Encrypted resources which have to be decrypted on startup

### Action Obfuscator

The second approach is to use encryption as obfuscation. The idea is to have a method, which is called by each method, to delegate all calls according to an encrypted parameter. The attacker can try to guess which method call is linked to each method, but when additional encryption for the parameters is applied and decryption is used in the target method, it is very hard to crack the logic without modifying a lot of code. In order to make it harder to circumvent the encryption of the methods, objects can be encrypted as well. An abstract presentation of the mechanism can be seen in figure 4.4.

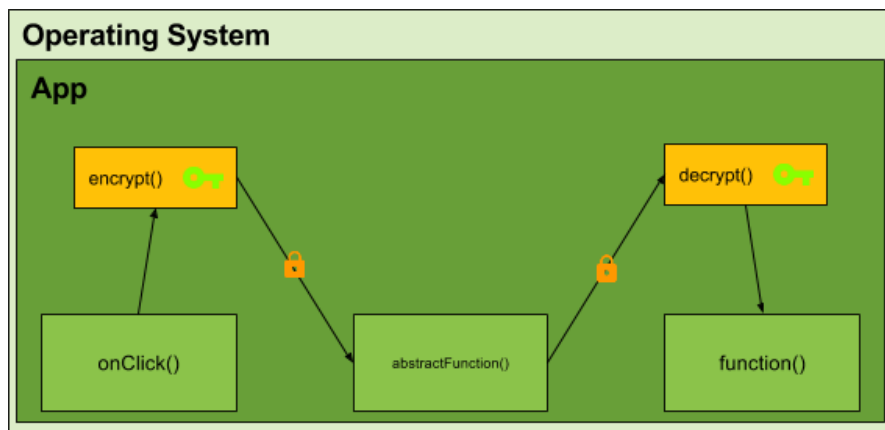


Figure 4.4: Encrypted actions to obfuscate dependencies

### Communication Decryption

The third approach is to use encryption on the server response as seen in figure 4.5. This is an additional security feature which is applied in combination with a content server described in subsection 4.2.1. When the user does the login on the server, additional unique device specific parameters have to be passed as well. On the first login, the server generates a cryptographic key which is used for communication with the user on this specific device. The corresponding key can either be generated on the device or be shared by the server. This mechanism allows only authorized users on a specific device to decrypt the communication.

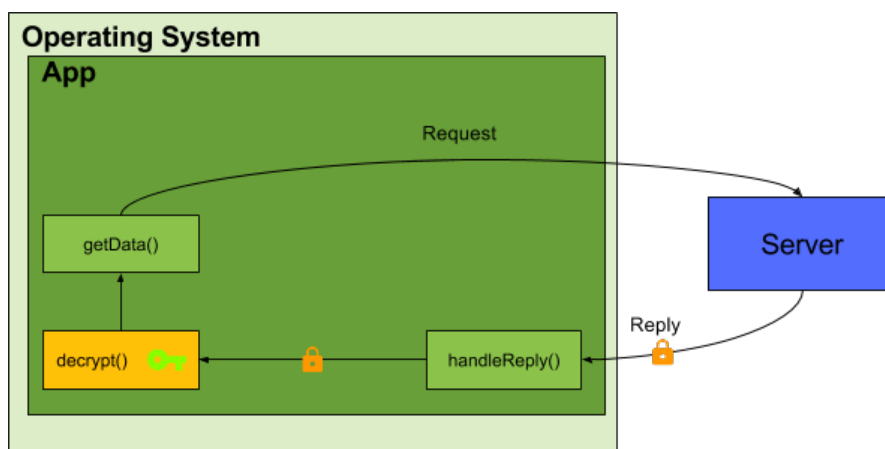


Figure 4.5: Encrypted communication with a server

A similar approach is used for streaming DRM protected content on Android. The encrypted content can only be decrypted by a native interface provided by the OS which stores the decryption key. [12]

This methodology focuses on the security of the content instead of the application itself.

### Encryption

After applying encryption on the application, the handling of the key has to be specified.

### Secure Element

An idea to handle the encryption key is to store it on a server and provide it to the application. This works similar to the license verification. When the decryption inside the application is called, the user is verified on the server. In case the check is successful, the decryption key is sent to the device. The advantage over having the original implementation is to have an unguessable result. The key can either be retrieved from the server for each decryption action or it can be cached on the device. Caching should be favored since getting the key for each action requires to be online, it slows down the application and it generates additional traffic. In order to improve security, keys can be changed when updating the version of the application.

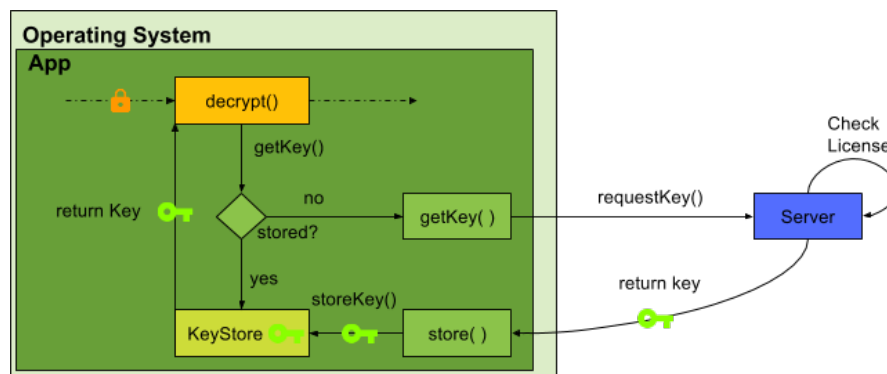


Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

### Secure Element

Since there is the possibility to read the cached encryption key [84] and crack the encryption, the use of Secure Element (SE) is proposed. A secure element is a tamper-resistant platform which can be used to securely host applications and cryptographic keys [48]. There are different form factors for SEs. For Android, the microSD form factor is the interesting one. It can be either mounted in the microSD card slot or by using an adapter on the USB interface, which requires the device to support USB On-The-Go (OTG) [usbOtg]. The resource is accessed over reads and writes to the filesystem. Since the SE has to be small to fit the size of a microSD card and powered by the host system, its hardware capabilities are constrained. The result is a performance of 25MHz which does not allow complex computations. [77]

For this reason the usage of the SE is restricted to simple tasks, like storing a key used for decryption. The advantage of an SE is that its functionality is outside of the Android



application and thus cannot be manipulated by Lucky Patcher.  
An abstract presentation of the use of a SE can be seen in figure 4.7.  
Integrating a SE comes with some problems.

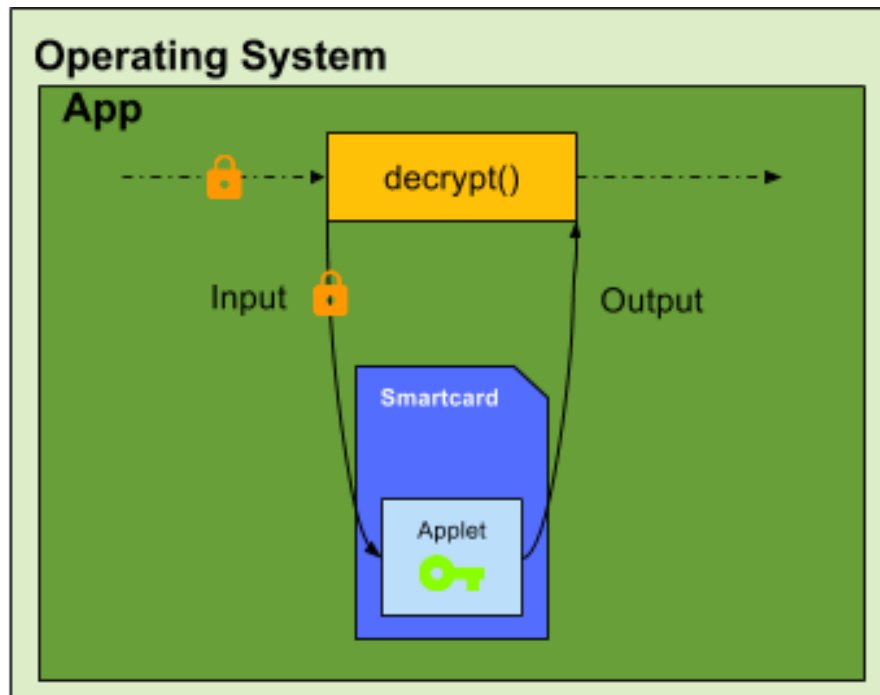


Figure 4.7: Decryption by using a smartcard

- the user has to buy extra hardware
- not all devices have a microSD card slot or support OTG
- different implementations for communication with the SE

The first problem is that the user is required to buy extra hardware. This means the user has to spend extra cash and to have the SE always around. The connection to the device using a cable is not the most convenient solution as well. The second problem is that not all devices either have an microSD card slot, nor OTG support. For example, the Nexus 7 (2012) and the Nexus 6P neither have the capability to use a microSD card. While the Nexus7 was supposed to have OTG, it did not work with the used SE, while the Nexus 6P did not support OTG out of the box. Both devices needed even needed additional plugins to read the OTG mounted microSD in a file

explorer. The third problem is that each manufacturer implements its own interpretation for the interface which makes SE incompatible to each other. For this reason, the SD Association proposed the smartSD in order to have a universal standard for SEs [74].

solution which are out there have major security flaws, smartcard itself can be attacked

have problems on their own, nur so sicher wie das secure element DAP Verification .... normalerweise muss jede Applet, die auf so ein Secure Element/Smartcard etc. kommt mit ner Signatur unterschrieben sein ...

Waehrend ich Exploits finden konnte, die Dir erw. Zugriff geben, wenn du Applets installieren kannst, u.a.

TODO: 2) Secure Elements Bottleneck ist sicherlich die Schnittstelle zu Android und alles was in Android ist, ist prinzipiell unsicher, also auch etwaige Keys. Was jedoch koennte Secure Elements absichern? Ich moechte dich bitten hier Ideen zu erarbeiten, was im Zuge von Kopierschutz, Verschluesslung etc. mit SEs wirklich sicher gemacht werden koennte. Eine grobe Idee ist z.B. das Signieren von Serveranfragen. Key kennt hier nur das SE und der Server. Android schickt die volle URL mit Parametern und das SE fuegt einen Signaturparameter zu. Vorteil: Ohne das SE kann die App den Server mal nicht mehr nutzen. Jetzt musste man verhindern, dass eine Proxy-App unter Android fuer andere aktiv wird (Stichwort CardSharing). Was koennte man tun? Das ist auch nur ein Idee. Was gibt es sonst noch? Wo koennte es Sinn machen einen sicheren Speicher zu haben?

Since Lucky Patcher focuses on the manipulation of the license verification libraries, it cannot be used for cracking encryption mechanisms. Thus implementing encryption protects from Lucky Patcher and instead the boundaries of encryption apply.

### 4.3 Additional Environmental Features

since dex code as we have seen is vulnerable to attacks and reengineering, the solution could be outside of the application, provided by either the environment or another hardware

level tiefer

#### 4.3.1 Trusted Execution Environment

This is my real text! Rest might be copied or not be checked!

WAS IST ES? WAS MACHT ES? WIE IMPLEMENTIERT MAN? <https://source.android.com/security/trusty/index.html> promoted as be all end all solution for mobile security in theory isolated processing core with isolated memory, cannot be

influenced by the outside and runs with privileged access allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of sensitive processing offloaded to protect information from malware

perfect wish: secure chip to process software that malware should not access, security related stuff like banking, encryption

example Trustzone, Knox

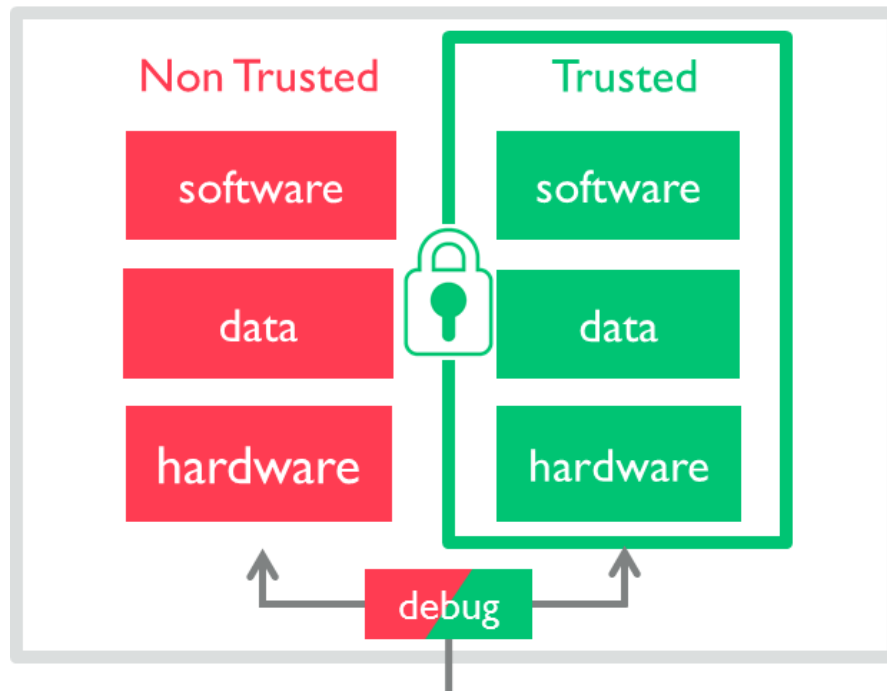


Figure 4.8: tee [29]

[34][29]

beispiele: new section trusted execution environment trusttronic letzte conference  
samsung knox

luckypatcher not able to attack because it cannot access the TEE but before using it as a safe solution some problems have to be fixed

e.g. trustzone what is it already used for secure data storage, hardware configurations, bootloader/sim lock no hide from malware but user

architecture problems kernel to your kernel trustzone image stored unencrypted physical memory pointers

protection validation can be done by either using qualcomm's or writing a custom one giant box, error by one has impact on all others

[34][29]

not accessible to anybody in general many different solutions, focus should be on one unique standard in order to fix the problems and make compatibility , google already started by integrating features of samsung's knox into android lollipop [71]

#### 4.3.2 ART

This is my real text! Rest might be copied or not be checked!

since dex is more like dangerous executable format and bears significant risks to app developers who do not use countermeasures against it

improve ART, already contains machine code which is hard to analyze and thus also difficult to find patches to apply with luckypatcher

already on the way, cannot be done from one day on the other, but right now not a protection against luckypatcher, will only be a solution when art code included in apks but why not now? Evaluation Why is Android not all ART now? Your applications still compile into Dalvik (DEX) code, Final compilation to ART occurs on the device, during install, Even ART binaries have Dalvik embedded in them, Some methods may be left as DEX, to be interpreted, Dalvik is much easier to debug than ART

[60]

zu ART. dex isnt dead yet, even with art still buried deep inside those oat files far easier to reverse engineer embedded dex than do so for oat

art is a far more advanced runtime architecture, brings android closer to ios native level performance vestiges of dex still remain to haunt performance, dex code is still 32 bit very much still a shifting landscape, internal structures keep on changing, google isnt afraid to break compatibility, llvm integration likely to only increase and improve for most users the change is smooth, better performance and power consumption, negligible cost binary size increase, minor limitations on dex obfuscation remain, for optimal performance and obfuscation nothing beats JNI

isn't android all dalvik now? art is runtime but application compile into dex, art is compiled on device during install, art binaries has dalvik embedded, some methods may be left as dex to be interpreted, dalvik is much easier to debug than art –see-evaluation

When creating odex on art it is directly put into art file

[60]

## 5 Conclusion

research and also a valuable market for companies

Because source code can be easier recovered from an application in comparison to x86, there is a strong need for code protection and adoption of existing reverse engineering methods. Main parts of Android application functionalities are realized in Dalvik bytecode. So Dalvik bytecode is of main interest for this topic

Also, the Android system does not prevent modification of this bytecode during runtime, This ability of modifying the code can be used to construct powerful code protection schemata and so make it hard to analyze a given application.

[73]

current state of license verification on Android reverse engineering far too easy due to OS, extract/install allowed gaining root easy, allows everyone especially pirates avoiding protection mechanisms java was chosen to support a lot of hardware, java has bad protection

lvl popular but broken, has not done much since beginning of known issues [64]

auch wichtig weil wenn crackable dann upload zu stores und dann malware

<http://www.hotforsecurity.com/blog/mobile-app-development-company-fights-off-android-malware-with-obfuscation-tool-3717.html>

### 5.1 Summary

jedes chapter beschreiben

### 5.2 Discussion

clear in beginnign that lvl not sufficiently safe with current technology unclear degree and fixavle

shortly after start insufficient reilience against reverse engineering, not explusivly to lvl thus shift from lvl protection to general protection against reverse engineering, decompilation and patching

eternal arms race no winning solution against all cases, just small pieces quantitative improvement no qualitatively improve resilience limited to quantitative resilience, matter of time until small steps generate more work for reengineering, ggf lower motivation for cracker only matter of time until patching tools catch up, completely new protection schemes need to be devised to counter those [64]

not a question of if but of when bytecode tool to generate the licens elibrary on the fly, using random permutations and injecting it everywhere into the bytecode with an open platform we have to accept a crack will happen [55]

um das ganze zu umgehen content driven, a la spotify, jedoch ist dies nicht mit jeder geschäftsidee machbar

alles hilft gegen lucky patcher auf den ersten blick, jedoch custom patches, welche Lucky Patcher anbietet[64], können es einfach umgehen, deswegen hilft nur reengineering schwerer zu machen viele piraten sind nicht mehr motiviert wenn es zu schwer ist every new layer of obfuscation/modification adds another level complexity

solange keine bessere lösung vorhanden unique machen um custom analysis und reengineering zu enforcen und dann viele kleine teile um die schwierigkeit des reengineeren und angriffs zu erschweren und viel zeit in anspruch zu nehmen um die motivation der angreifer zu verringern und somit die app zu schützen

### 5.3 Future Work

This is my real text! Rest might be copied or not be checked!

lvl has room for improvement art promising but not root issue, dex is distributed and art compilation to native on device needs to become relevant so developers can release art only apps, native code and no issue with reverse engineering stop/less important until lvl see major update custom improvements have to be done [64]

nicht mehr zu rettendes model, dex hat zu viele probleme, google bzw die andern anbieter müssen eine uber lösung liefern denn für den einzelnen entwickler so etwas zu ertellen ist nicht feasible, da einen mechanismus zu erstellen komplexer ist als die app itself

se/tee muss es eine lösung geben sonst braucht man für verschiedene apps verschiedene se, gemeinsame kraft um die eine lösung zu verbessern und nicht lauter schweizer käse zu ahben

google hat schon sowas wie google vault

all papers with malware and copyright protection is interesting since they also want to hide their code

## List of Figures

2.1	Different ways to generate revenue and how the pirate can cut them . .	7
2.2	Android's architecture [65] . . . . .	9
2.3	APK build process [60] . . . . .	11
2.4	APK folder structure . . . . .	12
2.5	.jar to APK transformation [31] . . . . .	13
2.6	.dex file format [60] . . . . .	14
2.7	Installing an APK on a device [26] . . . . .	15
2.8	artarch . . . . .	17
2.9	Timeline of market share of evolving Android versions and dates of root exploits recorded [45] [1] [41] [42] . . . . .	18
2.10	Google's implementation of license checking [19] . . . . .	21
2.11	Amazon library structure in decompiled application . . . . .	23
2.12	Developer preferences in the Amazon developer console [6] . . . . .	24
2.13	Abstraction of the current license verification mechanism. The library is represented by (1) . . . . .	27
2.14	Java .class and .dex can be transformed bidirectional [60] . . . . .	28
2.15	Overview of build and reverse engineering of an APK [60] . . . . .	29
3.1	Left to right: Features offered LuckyPatcher, modes to crack license verification and the result after patching . . . . .	39
3.2	Abstraction of the current attack on the license verification mechanism	54
4.1	Introduction of additional tests to check environment and integrity of the application . . . . .	59
4.2	Abstraction of an application and a content server . . . . .	68
4.3	Encrypted resources which have to be decrypted on startup . . . . .	70
4.4	Encrypted actions to obfuscate dependencies . . . . .	70
4.5	Encrypted communication with a server . . . . .	71
4.6	Retrieving the key after successful identification from the server and store it local on device . . . . .	72
4.7	Decryption by using a smartcard . . . . .	73
4.8	tee [29] . . . . .	75

## List of Tables

3.1	Patching patterns applied by each mode . . . . .	50
3.2	Functionality for the test apps before and after patching . . . . .	53



## List of Code Snippets

2.1	Include permission to check the license in AndroidManifest.xml [13] . .	22
2.2	Setting up the LVL license check call . . . . .	22
2.3	LVL license check callback . . . . .	23
2.4	Amazon's onCreate() injection to call Kiwi license verification as well .	25
2.5	Setting up the Zirconia license check call . . . . .	26
2.6	Zirconia license check callback . . . . .	26
2.7	Script to extract the .dex bytecode from the APK . . . . .	31
2.8	dex hexdump example . . . . .	31
2.9	Script to generate the corresponding smali code for a given APK . . . .	33
2.10	smali example . . . . .	33
2.11	Script to decompile to Java using androguard . . . . .	35
2.12	Java code example using androguard . . . . .	35
2.13	Script to decompile to Java using JADX . . . . .	35
2.14	Java code example using JADX . . . . .	36
2.15	Script to compare the original and manipulated APK to see the modifi- cations in the different presentations . . . . .	36
2.16	Diff on Dex level for N1 pattern . . . . .	37
3.1	Diff on Dex level for N1 pattern . . . . .	43
3.2	Diff on Smali level for N1 pattern . . . . .	43
3.3	Diff on Java level for N1 pattern (abstracted) . . . . .	44
3.4	Diff on Dex level for N2 pattern . . . . .	45
3.5	Diff on Smali level for n2 pattern . . . . .	45
3.6	Diff on Java level for N2 pattern (abstracted) . . . . .	45
3.7	Diff on Dex level for N3 pattern . . . . .	46
3.8	Diff on Smali level for N3 pattern . . . . .	46
3.9	Diff on Java level for N3 pattern (abstracted) . . . . .	47
3.10	Diff on Dex level for N4 patch . . . . .	47
3.11	Diff on Smali level for N4 patch . . . . .	47
3.12	Diff on Java level for N4 patch (abstracted) . . . . .	47
3.13	Diff on Java level for N5 patch (abstracted) . . . . .	48
3.14	Diff on Dex level for N6 patch . . . . .	48

3.15	Diff on Smali level for N6 patch . . . . .	49
3.16	Diff on Java level for N6 patch (abstracted) . . . . .	49
3.17	Diff on Java level for N7 patch (abstracted) . . . . .	50
3.18	Diff on Dex level for Amazon patch . . . . .	51
3.19	Diff on Smali level for Amazon patch . . . . .	51
3.20	Diff on Java level for Amazon patch (abstracted) . . . . .	51
3.21	Diff on Dex level for Samsung patch . . . . .	52
3.22	Diff on Smali level for Samsung patch . . . . .	52
3.23	Diff on Java level for Samsung patch (abstracted) . . . . .	53
4.1	Example code for checking for debuggability . . . . .	60
4.2	Example code for checking for root . . . . .	61
4.3	Example code for checking whether Lucky Patcher is installed on the device . . . . .	62
4.4	Example code for checking the origin of the installation . . . . .	63
4.5	Example code for checking the signature of the application . . . . .	64

# Bibliography

- [1] Alastair Beresford, et al. *All Vulnerabilities*. URL: <http://androidvulnerabilities.org/all> (visited on 01/24/2016).
- [2] allatori. *Allatori Java Obfuscator*. URL: <http://www.allatori.com/> (visited on 01/22/2016).
- [3] allatori. *Documentation*. URL: <http://www.allatori.com/doc.html> (visited on 01/22/2016).
- [4] allatori. *ProGuard*. URL: <http://developer.android.com/tools/help/proguard.html> (visited on 01/22/2016).
- [5] Almalence Inc. *A Better Camera*. URL: <http://www.amazon.de/Almalence-Inc-A-Better-Camera/dp/B00HUP8UZA> (visited on 02/17/2016).
- [6] Amazon. *Amazon Developer Service*. URL: <https://developer.amazon.com/> (visited on 02/02/2016).
- [7] Amazon. *Amazon Send Developers a Welcome Package*. URL: <http://www.androidheadlines.com/2010/10/amazon-send-developers-a-welcome-package.html> (visited on 01/19/2016).
- [8] Amazon. *Introducing Amazon Appstore for Android*. URL: <http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1541548> (visited on 01/19/2016).
- [9] Android. *ART and Dalvik*. URL: <https://source.android.com/devices/tech/dalvik/index.html> (visited on 02/15/2016).
- [10] Android. *Configuring ART*. URL: <https://source.android.com/devices/tech/dalvik/configure.html> (visited on 02/15/2016).
- [11] Android. *Dalvik bytecode*. URL: <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html> (visited on 02/29/2016).
- [12] Android. *DRM*. URL: <https://source.android.com/devices/drm.html> (visited on 02/29/2016).
- [13] Android Developers. *Adding Licensing to Your App*. URL: <https://developer.android.com/google/play/licensing/adding-licensing.html> (visited on 01/18/2016).

- [14] Android Developers. *Android NDK*. URL: <http://developer.android.com/tools/sdk/ndk/index.html> (visited on 02/21/2016).
- [15] Android Developers. *Application Fundamentals*. URL: <http://developer.android.com/guide/components/fundamentals.html> (visited on 01/18/2016).
- [16] Android Developers. *Checking Device Compatibility with SafetyNet*. URL: <https://developer.android.com/training/safetynet/index.html> (visited on 02/21/2016).
- [17] Android Developers. *Dalvik Executable format*. URL: <https://source.android.com/devices/tech/dalvik/dex-format.html> (visited on 02/02/2016).
- [18] Android Developers. *Debugging*. URL: <http://developer.android.com/tools/debugging/index.html> (visited on 02/21/2016).
- [19] Android Developers. *Licensing Overview*. URL: <https://developer.android.com/google/play/licensing/overview.html> (visited on 01/18/2016).
- [20] Android Developers. *Licensing Reference*. URL: <https://developer.android.com/google/play/licensing/licensing-reference.html> (visited on 01/21/2016).
- [21] Android Developers. *Setting Up for Licensing*. URL: <https://developer.android.com/google/play/licensing/setting-up.html> (visited on 01/18/2016).
- [22] Android Developers. *Settings.Secure*. URL: <http://developer.android.com/reference/android/provider/Settings.Secure.html> (visited on 02/16/2016).
- [23] Android Developers. *Signing Your Applications*. URL: <http://developer.android.com/tools/publishing/app-signing.html> (visited on 03/01/2016).
- [24] Android Developers Blog. *Announcing the Android 1.0 SDK, release 1*. URL: <http://android-developers.blogspot.de/2008/09/announcing-android-10-sdk-release-1.html> (visited on 02/15/2016).
- [25] AnjLab. *A lightweight implementation of Android In-app Billing Version 3*. URL: <https://github.com/anjlab/android-inapp-billing-v3> (visited on 02/18/2016).
- [26] I. R. Anwar Ghuloum Brian Carlstrom. *The ART runtime*. URL: <https://www.youtube.com/watch?v=EB1TzQsUo0w> (visited on 02/02/2016).
- [27] APKSFREE.com. *diff*. URL: <http://www.androidapkfree.com/app/blackmart-alpha-latest-version/> (visited on 02/11/2016).
- [28] Apple. *Piracy Prevention*. URL: <http://www.apple.com/legal/intellectual-property/piracy.html> (visited on 01/18/2016).
- [29] ARM. *TrustZone*. URL: <http://www.arm.com/products/processors/technologies/trustzone/index.php> (visited on 01/24/2016).

- [30] Blackmart. *Blackmart Alpha*. URL: <http://www.blackmart.us/> (visited on 01/20/2016).
- [31] D. Bornstein. *Dalvik VM Internals*. URL: <https://sites.google.com/site/io/dalvik-vm-internals> (visited on 02/02/2016).
- [32] L. Botezatu. *Manipulation und Diebstahl im Google Play Store*. URL: <http://www.bitdefender.de/hotforsecurity/manipulation-und-diebstahl-im-google-play-store-2673.html> (visited on 01/16/2016).
- [33] J. Callaham. *Smartphone OS Market Share*. URL: <http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide> (visited on 01/16/2016).
- [34] J. T. Charles Holmes. *An Infestation of Dragons - Exploring Vulnerabilities in the ARM TrustZone Architecture*. URL: [https://usmile.at/sites/default/files/androidsecuritysymposium/presentations/Thomas\\_Holmes\\_AnInfestationOfDragons.pdf](https://usmile.at/sites/default/files/androidsecuritysymposium/presentations/Thomas_Holmes_AnInfestationOfDragons.pdf) (visited on 01/24/2016).
- [35] ChelpuS. *Lucky Patcher*. URL: <http://lucky-patcher.netbew.com/> (visited on 01/09/2016).
- [36] W. Choi. *A mini project to customize existing AXML library*. URL: <https://github.com/wtchoi/axml> (visited on 02/17/2016).
- [37] E. Chu. *Licensing Service For Android Applications*. URL: <http://android-developers.blogspot.de/2010/07/licensing-service-for-android.html> (visited on 01/18/2016).
- [38] comScore. *comScore Reports November 2015 U.S. Smartphone Subscriber Market Share*. URL: <https://www.comscore.com/ger/Insights/Market-Rankings/comScore-Reports-November-2015-US-Smartphone-Subscriber-Market-Share> (visited on 01/19/2016).
- [39] ContentGuard. *AntiPiracySupport*. URL: <https://github.com/ContentGuard/AntiPiracySupport> (visited on 02/21/2016).
- [40] CrackAPK. *Android APK Cracked*. URL: <http://www.crackapk.com/> (visited on 01/20/2016).
- [41] CVE. *Common Vulnerabilities and Exposures*. URL: <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=android,+privileges> (visited on 02/25/2016).
- [42] CVE Details. *Google Android: List of Security Vulnerabilities (Gain Privilege)*. URL: [http://www.cvedetails.com/vulnerability-list/vendor\\_id-1224/product\\_id-19997/opgpriv-1/Google-Android.html](http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/opgpriv-1/Google-Android.html) (visited on 02/29/2016).

- [43] M. Dziatkiewicz. *Preventing Android applications piracy possible, requires diligence, planning*. URL: <http://www.fiercedeveloper.com/story/preventing-android-applications-piracy-possible-requires-diligence-planning/2012-08-14> (visited on 01/26/2016).
- [44] D. Ehringer. *The Dalvik Virtual Machine Architecture*. Mar. 2010.
- [45] Erikrespo. *Häufigkeit der verschiedenen Android-Versionen. Alle Versionen älter als 4.0 sind fast verschwunden*. URL: [https://de.wikipedia.org/wiki/Android\\_\(Betriebssystem\)#/media/File:Android\\_historical\\_version\\_distribution\\_-\\_vector.svg](https://de.wikipedia.org/wiki/Android_(Betriebssystem)#/media/File:Android_historical_version_distribution_-_vector.svg) (visited on 02/25/2016).
- [46] D. Galpin. *Proguard, Android, and the Licensing Server*. URL: <http://android-developers.blogspot.de/2010/09/proguard-android-and-licensing-server.html> (visited on 01/22/2016).
- [47] GitHub. *Atom - A hackable text editor for the 21st Century*. URL: <https://atom.io/> (visited on 02/17/2016).
- [48] GlobalPlatform. *GlobalPlatform made simple guide: Secure Element*. URL: <https://www.globalplatform.org/mediaguideSE.asp> (visited on 02/29/2016).
- [49] Google Developers. *AdMob for Android*. URL: <https://developers.google.com/admob/android/quick-start> (visited on 01/26/2016).
- [50] Google Play. *Google Play*. URL: <https://play.google.com/store?hl=de> (visited on 01/26/2016).
- [51] GuardSquare. *DexGuard - The strongest Android obfuscator, protector, and optimizer*. URL: <https://www.guardsquare.com/dexguard> (visited on 01/22/2016).
- [52] IDC Research, Inc. *Smartphone OS Market Share*. URL: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp> (visited on 01/16/2016).
- [53] T. Johns. *Securing Android LVL Applications*. URL: <http://android-developers.blogspot.de/2010/09/securing-android-lvl-applications.html> (visited on 01/18/2016).
- [54] E. Johnston. *Mobile Game Piracy Isn't All Bad, Says Monument Valley Producer (Q&A)*. URL: <http://recode.net/2015/01/06/mobile-game-piracy-isnt-all-bad-says-monument-valley-producer-qa/> (visited on 01/18/2016).
- [55] Kevin. *How the Android License Verification Library is Lulling You into a False Sense of Security*. URL: <http://www.digipom.com/how-the-android-license-verification-library-is-lulling-you-into-a-false-sense-of-security/> (visited on 01/18/2016).

- [56] A. Kovacheva. "Efficient Code Obfuscation for Android." Master's Thesis. Université de Luxembourg, Faculty of Science, Technology and Communication, Aug. 2013.
- [57] J. Kozyrakis. *AntiPiracySupport*. URL: <https://koz.io/inside-safetynet/> (visited on 02/21/2016).
- [58] M. Kroker. *App-Markt in Deutschland 2014: Umsätze im Google Play Store erstmals größer als bei Apple*. URL: <http://blog.wiwo.de/look-at-it/2015/02/25/app-markt-in-deutschland-2014-umsatze-im-google-play-store-erstmalsgroesser-als-bei-apple/> (visited on 01/16/2016).
- [59] J. Levin. *Android Security - New threats, New Capabilities*. URL: <http://newandroidbook.com/files/Andevcon-Sec.pdf> and (visited on 01/18/2016).
- [60] J. Levin. *Dalvik and ART*. Dec. 2015.
- [61] M. Liersch. *Android Piracy*. URL: <https://www.youtube.com/watch?v=TNnccRimhsI> (visited on 01/22/2016).
- [62] S. R. Lingala. *Zip4j - Java library to handle ZIP files*. URL: <http://www.lingala.net/zip4j/> (visited on 02/17/2016).
- [63] S. Morrow. *Rooting Explained + Top 5 Benefits Of Rooting Your Android Phone*. URL: <http://www.androidpolice.com/2010/04/15/rooting-explained-top-5-benefits-of-rooting-your-android-phone/> (visited on 01/18/2016).
- [64] M.-N. Muntean. "Improving License Verification in Android." Master's Thesis. Technische Universität München, Fakultät für Informatik, May 2014.
- [65] Obscure - community site theme. *Understanding the Android software stack*. URL: <http://maat-portfolio.mut.ac.th/~r4140027/?p=116> (visited on 01/27/2016).
- [66] Oracle. *JAR File Specification*. URL: <http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html> (visited on 02/15/2016).
- [67] G. Paller. *Dalvik opcodes*. URL: [http://pallergabor.uw.hu/androidblog/dalvik\\_opcodes.html](http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html) (visited on 02/18/2016).
- [68] R. Price. *DexFile*. URL: <http://www.businessinsider.com/android-app-profitability-v-ios-2015-1?IR=T> (visited on 01/16/2016).
- [69] Pulser\_G2. *A Look at Marshmallow Root and Verity Complications*. URL: <http://www.xda-developers.com/a-look-at-marshmallow-root-verity-complications/> (visited on 02/22/2016).
- [70] Runtastic. *Runtastic PRO Laufen & Fitness*. URL: <https://play.google.com/store/apps/details?id=com.runtastic.android.pro2&hl=de> (visited on 01/20/2016).

- [71] Samsung. *A closer look at KNOX contributing in Android*. URL: <https://www.samsungknox.com/en/androidworkwithknox> (visited on 01/24/2016).
- [72] Samsung. *How to protect your app from illegal copy using Samsung Application License Management (Zirconia)*. URL: <http://developer.samsung.com/technical-doc/view.do?v=T0000000062L> (visited on 01/19/2016).
- [73] P. Schulz. "Code Protection in Android." Lab Course. Friedrich-Wilhelms-Universität Bonn, Institute of Computer Science, July 2012.
- [74] SD Association. *smartSD Memory Cards*. URL: <https://www.sdcard.org/developers/overview/ASSD/smartsd/> (visited on 02/29/2016).
- [75] M. T. Serrafiero. *Piracy Testimonies, Causes and Prevention*. URL: <http://www.xda-developers.com/piracy-testimonies-causes-and-prevention/> (visited on 01/16/2016).
- [76] Spotify Ltd. *Spotify Music*. URL: <https://play.google.com/store/apps/details?id=com.spotify.music&hl=de> (visited on 02/21/2016).
- [77] ST life.augmented. *Allatori Java Obfuscator*. URL: <http://www.st.com/web/catalog/mmc/FM143/SC1282/PF259413> (visited on 01/24/2016).
- [78] stack overflow. *Posts containing 'Android, Piracy'*. URL: <http://stackoverflow.com/search?q=android+piracy> (visited on 01/26/2016).
- [79] stack overflow. *Posts containing 'Lucky Patcher'*. URL: <http://stackoverflow.com/search?q=lucky+patcher> (visited on 01/26/2016).
- [80] statista. *Number of apps available in leading app stores as of July 2015*. URL: <https://its.uncg.edu/Software/Licensing/> (visited on 01/16/2016).
- [81] P. Steinlechner. *Cracker beißen sich die Zähne an "Just Cause 3" aus*. URL: <http://www.sueddeutsche.de/digital/illegale-kopien-von-computerspielen-cracker-beissen-sich-die-zaehne-an-just-cause-aus-1.2810482> (visited on 01/26/2016).
- [82] Stericson. *Busybox - Android-Apps auf Google Play*. URL: <http://www.androidheadlines.com/2010/10/amazon-send-developers-a-welcome-package.html> (visited on 01/19/2016).
- [83] TeamSpeak Systems GmbH. *TeamSpeak 3*. URL: <https://play.google.com/store/apps/details?id=com.teamspeak.ts3client&hl=de> (visited on 01/20/2016).
- [84] P. Teoh. *How to dump memory of any running processes in Android (rooted)*. URL: <https://tthtlc.wordpress.com/2011/12/10/how-to-dump-memory-of-any-running-processes-in-android-2/> (visited on 02/29/2016).



- [85] The University of North Carolina Greensboro. *Software Licensing*. URL: <http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/> (visited on 01/16/2016).
- [86] J. S. Tim Strazzare. *Android Hacker Protection Level 0*. URL: <https://www.youtube.com/watch?v=6vFcEJ2jg0w> (visited on 01/22/2016).
- [87] ubuntuusers. *diff*. URL: <https://wiki.ubuntuusers.de/diff> (visited on 02/02/2016).
- [88] J. Underwood. *Today Calendar's Piracy Rate*. URL: <https://plus.google.com/+JackUnderwood/posts/jWs84EPNyNS> (visited on 01/16/2016).
- [89] WugFresh. *Nexus Root Toolkit v2.1.4*. URL: <http://www.wugfresh.com/nrt/> (visited on 02/16/2016).
- [90] Za hiD. *ANTI-PIRACY SOFTWARE ACTIVATED SOLVED*. URL: <http://android-onex.blogspot.de/2015/07/anti-piracy-software-activated-solved.html> (visited on 02/21/2016).