DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Counter Measurements 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 Counter Measurements 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

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

Acknowledgments **iii**

Assumptions **iv**

Abstract **v**

Glossary **1**

Acronyms **2**

1 Introduction **4**
- 1.1 Licensing 4
- 1.2 Motivation 4

2 Foundation **6**
- 2.1 Software Piracy 6
  - 2.1.1 Threat to Developers 6
  - 2.1.2 Risks to Users 7
  - 2.1.3 Piracy on Android 7
- 2.2 Android 8
  - 2.2.1 Android Application Package (APK) 9
  - 2.2.2 Dalvik Executionable File Format 11
  - 2.2.3 Overview of Installation 13
  - 2.2.4 Dalvik Virtual Machine 14
  - 2.2.5 Android Runtime 14
  - 2.2.6 Copy Protection and Root 17
- 2.3 License Verification Libraries 19
  - 2.3.1 Google's License Verification Library 20
  - 2.3.2 Amazon DRM 23
  - 2.3.3 Samsung DRM 25
- 2.4 Code Analysis 26

3 Cracking Android Applications with LuckyPatcher **35**
- 3.1 What is LuckyPatcher and what is it used for? 35

3.2 Modus Operandi 36
3.3 Variants for Cracking License Verification 38
3.4 Patching Patterns 38
3.5 Learnings from LuckyPatcher 46

4 Counter Measurements for Developers **48**
4.1 Tampering Protection 48
4.1.1 Prevent Debuggability 49
4.1.2 Root Detection 50
4.1.3 LuckyPatcher Detection 51
4.1.4 Sideload Detection 52
4.1.5 Signature 53
4.1.6 Flow Control 54
4.2 Library Modifications 55
4.2.1 Modify the Library 55
4.2.2 Native Implementation 56
4.3 Reverse Engineering Prevention 56
4.3.1 Break Common Reengineering Tools 57
4.3.2 Obfuscation 58
4.3.3 Packers 61
4.4 External Support 62
4.4.1 Service-managed Accounts 63
4.4.2 ART endlich durchsetzen 63
4.4.3 Secure Elements 63
4.4.4 Trusted Execution Environment 63

5 Evaluation of Counter Measurements **65**
5.1 Tampering Protection 65
5.2 Library Modifications 65
5.2.1 Modify the Library 65
5.2.2 Native Implementation 66
5.3 Reverse Engineering Prevention 67
5.3.1 Break Common Reengineering Tools 67
5.3.2 Obfuscation 68
5.3.3 Packers 71
5.3.4 Packers 73
5.4 External Support 73
5.4.1 Service-managed Accounts 73
5.4.2 ART endlich durchsetzen 74

5.4.3 Secure Elements 74
5.4.4 Trusted Execution Environment 74

6 Conclusion **76**
6.1 Summary 76
6.2 Discussion 76
6.3 Future Work 77

List of Figures **79**

List of Tables **80**

List of Code Snippets **81**

Bibliography **83**

Glossary

.class Java Byte Code produced by the Java compiler from a .java file.

.dex Dalvik Byte Code 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 Byte Code 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..

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.

APK 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.

NDK Native Development Kit.

OS operating system.

SDK Software Development Kit.

TUM Technische Universität München.

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. On the one hand, the goal is to protect the software creator's intellectual property (IP) or other features and enable him to commercialize it as a product. On the other hand it defines the boundaries for the user and prevents him from illicit usage [49].

Since licensing software often includes fees, these products often have mechanisms implemented to enforce the legal agreement and to prevent unauthorized use. The solutions range from providing limited features or usage for a limited time to disabling the software in case no legit account or serial key is provided.

But as long as there have been these protection mechanisms there have been software pirates who tried to circumvent it. It is an everlasting arms race[50].

1.2 Motivation

According to Google's in September 2015 there were over 1.4 billion active devices in the last 30 days and a market share of almost 82.8% in Q2 of 2015 [13][27]. This giant number of Android devices is powered by Google Play [24], Google's marketplace. It offers different kinds of digital goods, as movies, music or ebooks, but also hardware. In the application section of Google Play user can chose from over 1.6 million applications for Android [57]. In 2014 Google's marketplace overtook Apple's Appstore, which had a revenue of over 10 billion in 2013, and became the biggest application store on a mobile platform [32].
The growth has many advantages. Some time ago developers only considered iOS user as profitable and thus most applications were developed for Apple's operating system (OS). Now with this reach and potential users they also focus on Android[41]. But there are downsides as well. The expanding market for Android, offering many high

quality applications, also draws the attention of software pirates. But Crackers do not only to bypass license mechanisms and offer applications for free. Redirect cash flow or distributing malware using plagiates is an lucrative business model as well. Android developers are aware of this situation [59] and express their need to protect their IP on platforms like xda-developers [45] or stackoverflow [47]. Many of the developers having problems with the license verification mechanism name Lucky Patcher as one of their biggest problems [48].

The scope of this thesis is to analyse Android cracking applications and to investigate in counter measurements for developers.

2 Foundation

Before understanding the attack mechanisms and discussing counter measurements, 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 and reengineering tools and methodologies.

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 [9]. It includes the not terms of use conform installation software by an individual as well as commercial resale of illegal software. Piracy is subject on all platforms and considered stealing.

2.1.1 Threat to Developers

Especially for software developers piracy is a problem. The most apparent issue is regarding lost revenue and clear at first glance.
When people are downloading an application for free and do not pay for it, there is no revenue generated and the developer does not earn any money.
At second glance, the facts are more complex. Income is not only lost when the user is not paying on purchase, the pirate can also modify the application to influence the follow up revenue. An example for the loss of future earnings is the unlocking of inapp purchases. Inapp purchases are a popular source of income for so called freemium games or lite versions of apps. The application itself is for free but has, for example, an ingame currency (freemium) respectively limited features (lite version). The revenue is generated by offering additional ingame currency (freemium) or the unlocking of all features (lite version) in exchange for money. When the inapp purchases are unlocked, the process executed with success but no payment is transfered. Thus no earnings are generated for the developer.
An different example for the redirection of revenue is the modification of the Ad Unit ID [23]. The Ad Unit ID is responsible to assign earnigns generated by an mobile advertisment to the developer. When an application is pirated, this code can be replaced

by the pirate's one. Future revenues generated by advertisments in the application will not be assigned to the developer but the pirate.

Additional problems arise when the app is taken from the environment of an official app store and moved to a blackmarket store or website. The first is the loss of control over the application. The developer can no longer provide support or updates for the application in case of malfunction. The second one is that users connect maliscious behaviour to the developer and not to the pirated version. Both these issues cause mistrust in the developer and might influence future revenues which are not even connected to this applciation. The third problem is unpredictable traffic. When distributing an application over Google Play the developer can monitor growth and adjust eventual servers accordingly. But when the application is distributed over unknown sources the developer cannot evaluate, the traffic can increase dramatically and cause a bad user experience for legitimate users as well [35].

Developers have to live of their applications. When they do not earn money, either because the revenue stream is redirected or because their IP is stolen and commercialized by someone else, they cannot continue with developing and their skills are lost.

2.1.2 Risks to Users

Pirated software is not only bad for developers but for users as well. Users use pirated applications because they seem to be free. The application might be altered in different ways, e.g. malware can be included, it steals personal data or has changed permissions. The user will not notice it right away since these "features" often happen in the background without the knowledge of the user, e.g. send SMS using an expensive service. Even if there is no malicious content implemented, the applications can suffer from bad stability due to removed license verification. In general the risk is very high that pirated software has a worse user experience than the original.[12][35]
In general pirated software should not be installed since it cannot be ensured without deep inspection that the application is doing what it is supposed to do.

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 [20]. Sources for pirated applications can be easily found on the internet. Simple searches containing "free apk" and the applications name return plenty of results on Google Search. The links direct to blackmarket applications, as Blackmart [11], and websites for cracked Android

Application Package (APK)s, as crackApk[18]. These providers claim to be user friendly because they offer older versions of applications or do not charge money for complete version of applications. In general they practice professional stealing and are dangerous for users. When downloading an APK it is not possible to guarantee the integrity of the program.
An example for the dimensions piracy can reach for a single application is "Today Calendar Pro". The developer stated in a Google+ post that the piracy rate of the application is as high as 85%. This results in only 15% being legitimately purchased and installed. [45][59]
For this reason some developers do not implement any copy protection at all since it is cracked within days [29]. Especially Android applications are at high risk for piracy due bytecode in general is an easy target to reverse engineer.

2.2 Android

Android is an open source mobile OS launched in 2007 and currently developed by Google. It is based on the Linux kernel and targets touch screen devices 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 since these processors with their low power consumption are often used in this scenario. The following will give an quick overview over the architecture of Android and a deeper insight in the runtime powering Android.

The architecture of the software stack of Android can be seen in figure 2.1.
The basis of the system is its kernel. The kernel is responsible for power and memory management and is capable to execute standard Unix commands. It provides an hardware abstraction layer for software and controls the device drivers.
The layer on top of the kernel contains Android RunTime (ART), which will be explained in detail, as well as the the native libraries of the system. Android libraries are usually written in Java except these native and speed critical ones. They are written in C or C++ and allow low level interaction between applications and the kernel by using the Java Native Interface (JNI). This includes libraries like OpenGL, multimedia playback or the SQLite database.
Above this layer is the application framework. The application framework provides generic functionality as notification support to applications over Androids Application Programming Interface (API).
Applications are installed and executed on the top layer.
This structure enables Android to be run on a wide range of devices.

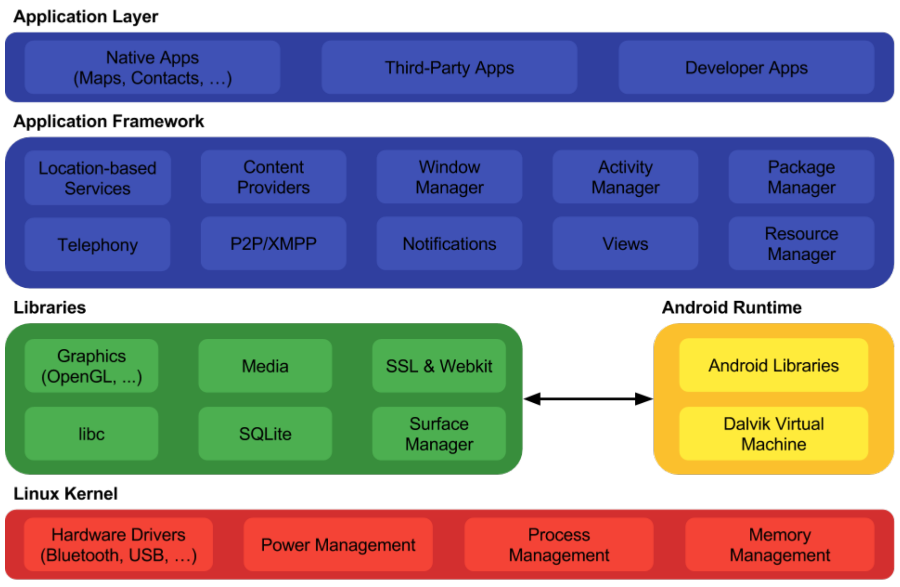


Figure 2.1: Android's architecture [39]

2.2.1 Android Application Package (APK)

Android applications are installed and distributed in the APK file format. They can either be installed from an appstore 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 resources of the application. The resources are added in the build process which is visualized in figure 2.2.
Android applications are usually written in Java.

step 1 Java files which will be compiled into .class files by a Java Compiler, similar to a Java program build process, class files contain Java bytecode representing the compiled application, optional step apply a Java Obfuscator
Standard Java environment compiles each separate class in the .java source code file into a corresponding Java bytecode .class file. Each class will be compiled into a single .class file. These are later packed together in a single .jar archive file. The JVM unpacks the .class files, parses and executes their code at runtime.

step 2: transformation from Java bytecode into Dalvik bytecode, see oben, dx programm from android sdk (due to it is necessity for building an application for the Android platform), output saved in singel .dex file, included in an APK in next step, possible to apply a further obfuscator operating on Dalvik bytecode(ERKLÄRUNG)

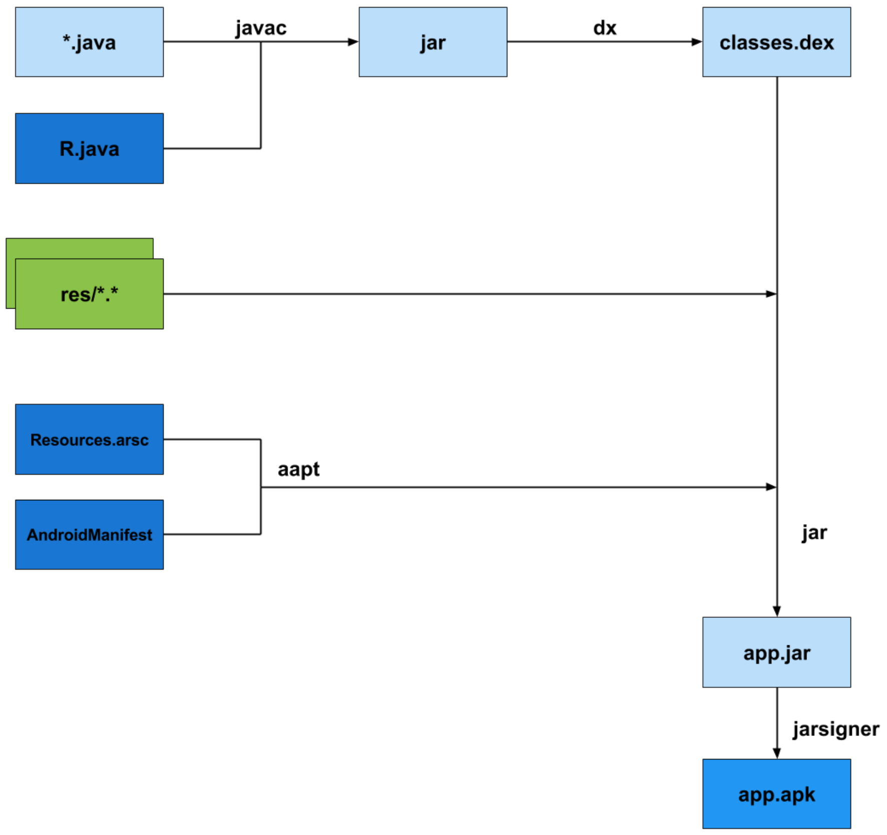


Figure 2.2: apk [34]

On the Android platform, the build process differs after the point when the .class files have been generated. Once having the latter, they are forwarded to the “dx” tool which is part of the standard Android SDK. This tool compresses all .class files into a single classes.dex file i.e. the .dex file is the sole bytecode container for all the application’s classes. After it has been created, the classes.dex is forwarded to the ApkBuilder tool altogether with the application resources and shared object (.so) files which, if present, contain native code. As a result, the APK archive is created and the final compulsory step is its signing. Figure 1.2 shows the APK build process and the possible obfuscation manipulations which are optional during the build stages

In the third step the ApkBuilder creates an APK which includes the classes.dex file and other resources. classes.dex: The classes compiled in the dex file format understandable by the Dalvik virtual machine

In the fourth and final step the jarsigner adds the developers signature to the package. lib: the directory containing the compiled code that is specific to a software layer of a processor, the directory is split into the different processor types res: the directory containing resources not compiled into resources.arsc (see below). assets: a directory containing applications assets, which can be retrieved by AssetManager. resources.arsc: a file containing precompiled resources, such as binary XML for example. AndroidManifest.xml: An additional Android manifest file, describing the name, version, access rights, referenced library files for the application. This file may be in Android binary XML that can be converted into human-readable plaintext XML with tools such as AXMLPrinter2, android-apktool, or Androguard. classes.dex: The classes compiled in the dex file format understandable by the Dalvik virtual machine

The signing does not improve security of the application itself but makes it possible to identify the developer and makes it possible to install updates for the application. META-INF directory: MANIFEST.MF: the Manifest file CERT.RSA: The certificate of the application. CERT.SF: The list of resources and SHA-1 digest of the corresponding lines in the MANIFEST.MF file

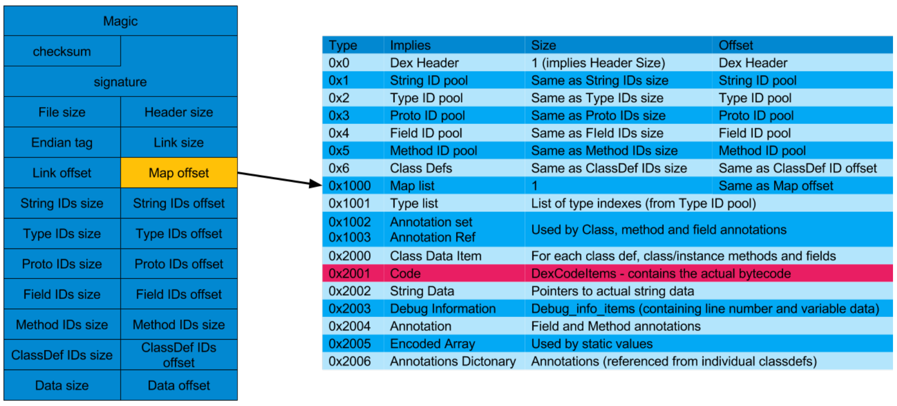
2.2.2 Dalvik Executionable File Format

Android applications deliver their code in Dalvik EXecutable (.dex) bytecode. Applications on Android are executed by the Dalvik Virtual Machine (DVM) in .dex bytecode. It can be compared to Java bytecode except some differences. While the Java virtual machine (VM) is stack-based the DVM is register-based, this circumstances have influence on the code. The Java bytecode is actually more compact than dex since it uses 8bit constants .dex which has instructions of 16bit multiples.

The bytecode is more suited to run on the ARM architecture since it supports direct mapping from dex registers to the registers of the ARM processor. Registers in .dex bytecode are 32bits wide and store values such as integers or float values. In case there are 64bit values, adjacent registers are used to store it. The .dex bytecode supports 218 valid opcodes which have a dest-sort ordering for its arguments. The instructions refer to indexes in pools.

The applications for Android are written using the Java programming language. The process can be seen in figure 2.2. The first steps are similar to the build process of Java applications. In the first step the Java files are compiled The Java files are compiled into .class files by the Java compiler step 1: Java files which will be compiled into .class files by a Java Compiler, similar to a Java program build process, class files contain Java bytecode representing the compiled application, optional step apply a Java Obfuscator

When compiling, the application is first compiled to Java Archive (.jar) file using the



Type	Implies	Size	Offset
0x0	Dex Header	1 (implies Header Size)	Dex Header
0x1	String ID pool	Same as String IDs size	String ID pool
0x2	Type ID pool	Same as Type IDs size	Type ID pool
0x3	Proto ID pool	Same as Proto IDs size	Proto ID pool
0x4	Field ID pool	Same as FIeld IDs size	Field ID pool
0x5	Method ID pool	Same as Method IDs size	Method ID pool
0x6	Class Defs	Same as ClassDef IDs size	Same as ClassDef ID offset
0x1000	Map list	1	Same as Map offset
0x1001	Type list	List of type indexes (from Type ID pool)	
0x1002 0x1003	Annotation set Annotation Ref	Used by Class, method and field annotations	
0x2000	Class Data Item	For each class def, class/instance methods and fields	
0x2001	Code	DexCodeItems - contains the actual bytecode	
0x2002	String Data	Pointers to actual string data	
0x2003	Debug Information	Debug_info_items (containing line number and variable data)	
0x2004	Annotation	Field and Method annotations	
0x2005	Encoded Array	Used by static values	
0x2006	Annotations Dictonary	Annotations (referenced from individual classdefs)	

Figure 2.3: dex [34]

Java compiler javac. The jar contains the .class, which are the Java classes in bytecode, as well as the heterogenous constant pool. In order to create the dex file, which is called classes.dex, dx is used on the .jar file. Dx compiles the Java bytecode to .dex byte code and sorts string, type and method from the heterogenous pool in seperated pools and removes dublicates. This results in figure 2.4 This is most effective for string and this action results in a memory footprint which is up to 44% lower than the one of the .jar. The result is that the .dex file has significant more references than the .jar file. The .dex file is stored as classes.[21] Like Java bytecode .dex bytecode allows easy decompilation

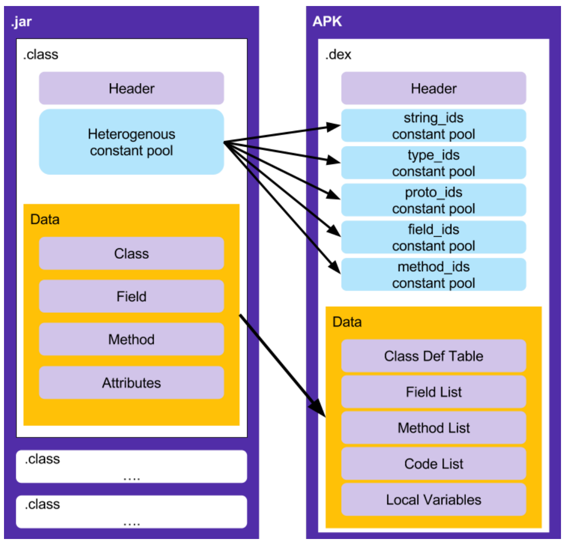


Figure 2.4: java

to Java. Also since the bytecode is in contrast to other architectures pretty simple and only in rare cases protection is applied, it is an easy target for reverse engineering.

Since .dex bytecode supports optimization, improvements for the underlying architecture can be applied to the bytecode upon installation. The resulting .dex file is called Optimized Dalvik EXecutable (.odex). The optimizations are executed by a programm called dexopt which is part of the Android plattform. For the DVM it makes no difference whether .dex or .odex files are executed, except speed improvements.

2.2.3 Overview of Installation

Installation two steps: primary is the APK verification and secondary is the bytecode optimization/art
legitimate signature as well as correct classes.dex structure cannot be verified are rejected for installation by the OS
Once verified, the .dex file is forwarded for optimization: a necessary step due to the high diversity of Android running hardware (dex)-see- Dalvik executable is a generic file format which needs additional processing to achieve best performance for the concrete device architecture (odex)
optimization
step removes the classes.dex from the original APK archive and loads in memory the .odex file upon execution, occurs only once, during the initial run of the application which explains the usually slower first application launch comparing to the subsequent ones[31]

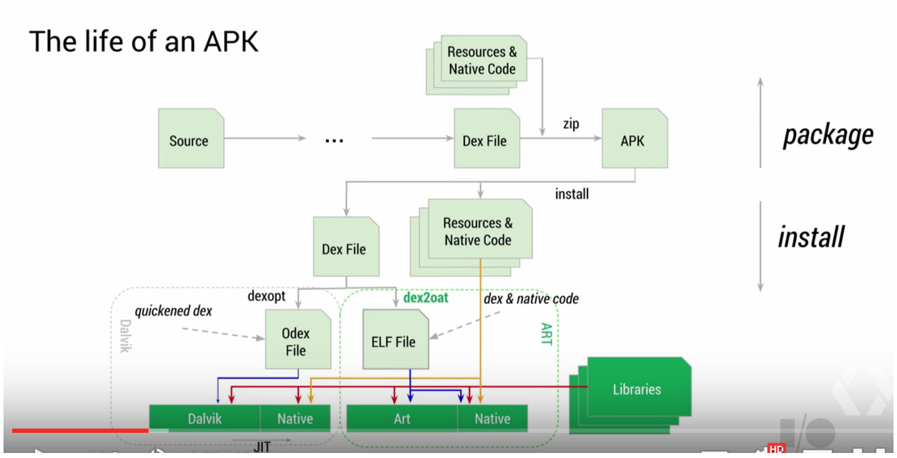


Figure 2.5: install

In order to create a sandboxed environment, Android assigns each application a seperate user ID on install to ensure that nobody except users with the same ID can access the applications resources. The Android operating system is a multi-user Linux system in which each app is a different user. By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps. By default, every app runs in its own Linux process. Android starts the process when any of the app's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other apps[7]

2.2.4 Dalvik Virtual Machine

The DVM, created by Dan Bornstein and named after an Iclandic town, was introduced along with Android.
In contrast to a stationary computer a mobile device have a lot of constraints. Since they are powered by battery the processing power and RAM are limited to fit power consumtion. In addition to these hardware limitations Andriod has some additional requirements, like no swap for the RAM, the need to run on a diverse set of devices and a sandboxed application runtime . In order to run efficient it has to be designed according to these requirements. The DVM a customized and optimized Java Virtual Machine (JVM) based on Apache Harmony and thus is related to 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 they need less instructions for execution than stack-based architecture which results in less CPU cycles. The downside is an approximatly 25% larger codebase and negligible larger fetching times [21]. In addition to the lower level changes the DVM is optimized for memory sharing, it stores references bitmaps seperately from objects and optimizes application startup by using zygotes [34].

The last change made to DVM was the introduction of Just-In-Time (JIT) in Android version 2.2 "Froyo".

2.2.5 Android Runtime

In Android version 4.4 "Kitkat" Google introduced ART which was optional and only available through the developer options. With the release of verison 5.0 "Lollipop" ART it became the runtime of choice since DVM had some major flaws.

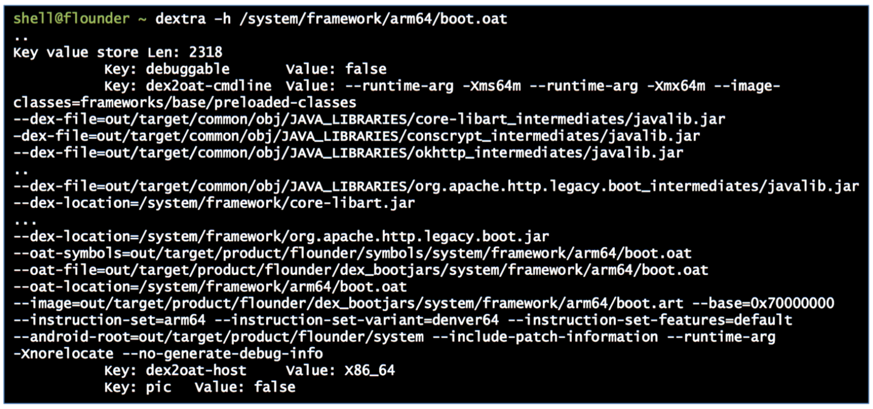
JIT is slow, consuming both cycles and battery power gabage collection causes hangs/jitters dalvik is 32bit, cannot beneift from 64bit architecture

art introduced in kitkat 4.4, available only through developer options, declared to be preview release, own risk, very little documentation, if any in lollipop art becomes runtime of choice, supersedes dalvik, breaks compatibility with older dex, as well as itself, very little docu constantly evolving through marshmallow, major caveat : oftenc changes in between android minro versions, android optimizes apps everytime you update
art was designed to address shortcomings of dalvik: virutal machine maintenance is expensie, interpreter/jit simply arent efficient as native code, doing jit all over again on every execution is wasteful, maintenance threads require significantly mroe cpu cycles, cpu cycles translate to slower performance and shorter battery life dalvik garbage collection frequently causes hangs/jitter virtual machine architecture is 32bit only, android i sfollowing ios into 64bit space
advantages of art art mvoes compilation from JIT to Ahead-Of-Time (AOT) VM maintenance not as expensive as dalvik, art compiles to native AOT not JIT, less maintenance threads and overhead cycles than dalvik garbage collection parallizable (foreground/background), non blocking -see- QUELLE 64bit bus some issues still exist -see- quelle
main idea of art/aot: actually compilation can be to one of two types, quick (native code), portable(llvm code) in practice preference is to compile to naitve code, portable implies another layer of IR (LLVM's bit code)
Art itself: art uses not one but two file formats art - only one file, boot.art, in /syste/framework/architecture (arm,...) oat - master file, boot.oat, in /syste/framework/architecture (arm,...) - odex files no longer optimized dex but oat, alongeside 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-initilited 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

art file format

the oat dexfile header oat headers are 1...n dex files, actual value given by dexfilecount field in header

finding dex in oat odex files will usually have only one (=original) dex embedded



```
shell@flounder ~ dextra -h /system/framework/arm64/boot.oat
..
Key value store Len: 2318
          Key: debuggable      Value: false
          Key: dex2oat-cmdline  Value: --runtime-arg -Xms64m --runtime-arg -Xmx64m --image-
classes=frameworks/base/preloaded-classes
--dex-file=out/target/common/obj/JAVA_LIBRARIES/core-libart_intermediates/javalib.jar
-dex-file=out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/javalib.jar
--dex-file=out/target/common/obj/JAVA_LIBRARIES/okhttp_intermediates/javalib.jar
..
--dex-file=out/target/common/obj/JAVA_LIBRARIES/org.apache.http.legacy.boot_intermediates/javalib.jar
--dex-location=/system/framework/core-libart.jar
...
--dex-location=/system/framework/org.apache.http.legacy.boot.jar
--oat-symbols=out/target/product/flounder/symbols/system/framework/arm64/boot.oat
--oat-file=out/target/product/flounder/dex_bootjars/system/framework/arm64/boot.oat
--oat-location=/system/framework/arm64/boot.oat
--image=out/target/product/flounder/dex_bootjars/system/framework/arm64/boot.art --base=0x70000000
--instruction-set=arm64 --instruction-set-variant=denver64 --instruction-set-features=default
--android-root=out/target/product/flounder/system --include-patch-information --runtime-arg
-Xnorelocate --no-generate-debug-info
          Key: dex2oat-host     Value: X86_64
          Key: pic   Value: false
```

Figure 2.6: oat

booat.oat is something else entirely, some 14 dex files the best of the android framework jars, each dex contains potentilly hundres of classes

art code generation oat method headers point to offset of native code each method has a quick or portable method header, contains mapping from virtual register to underlying machine registers each method has a quick or protable frame info, provides frame size in bytes, core register spill mask, fp register spill mask generated code uses unusual registers, especially fond of using lr as call register, still saves/restores registers so as not to violate arm conventions

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

vergleich java/dex/odex(art) code

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

ART runtime threads the daemon threads are started in java by libcore, daemon class wraps thread class and provides singleton INSTANCE, do same basic operations as they did in "classic" dalvikVM, libart subree in libcore implementation slightly different

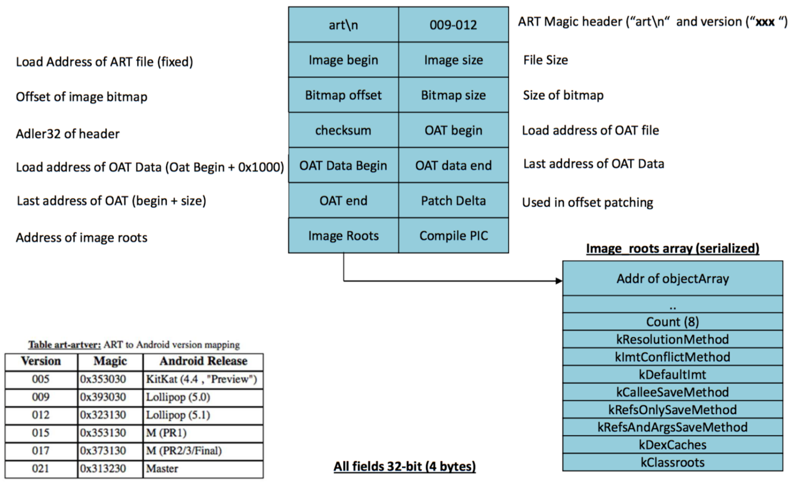


Table art-artver: ART to Android version mapping

Version	Magic	Android Release
005	0x353030	KitKat (4.4 , "Preview")
009	0x393030	Lollipop (5.0)
012	0x323130	Lollipop (5.1)
015	0x353130	M (PR1)
017	0x373130	M (PR2/3/Final)
021	0x313230	Master

Figure 2.7: art

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
[34]

2.2.6 Copy Protection and Root

unauthorized usage of an app through copy protection apk was installed in a location on the phone /data/app user could not access useless if single user can get apk and redistribute, gained by root successful as long as root was not easy for everyone -see-samsung rooting odin QUELLE, did not have too big impact as today, oneclick root kits QUELLE, standard for nexus

on install app's classes.dex is copies an optimized version (odex) to dalvik cache /data/dalvik-cache/, for faster startup (contains system apps and frameworks as well) odex erklären, byteswapping, structure realigning and memory-mapping when app is started optimized code from dalvik cache will be executed instead of apk old system was to put APK in folder user cannot access, unless you root weak, almost non-existant, anyone who can copy it can distribute it

It replaces the old system copy protection system, wherein your APKs would be put in a folder that you can't access. Unless you root. Oh, and anyone who can copy that APK off can then give it to someone else to put on their device, too. It was so weak, it

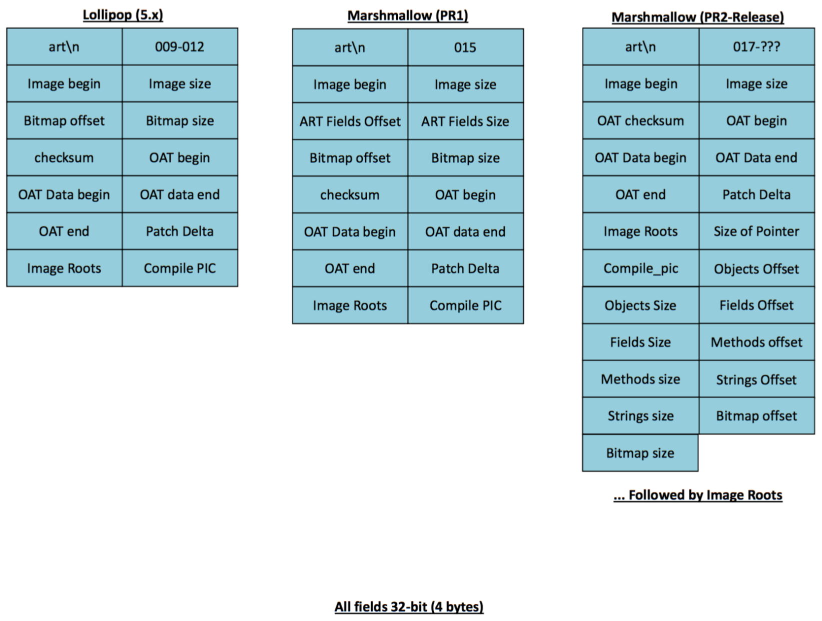


Figure 2.8: art2

was almost non-existant. kann mit root umgangen werden

Im original vom Markt direkt rutnergeladen und dann wird sie an den ort geschoben und kann nicht mehr zugegriffen werden -see- rechte etc, QUELLE

getting root/rooting process of modifying the operation system that shipped with your device to grant you complete control over it overcome limitations by carriers/-manufacturers, extend system functionality, upgrade to custom flavor root comes from Linux OS world, most privileges user on the system is called root rooting fairly simple, many videos and tutorials, sometimes oneclick tools not aproved by manufacturers or carriers, can not prevent usually exploits vulnerability in operating system code or device drivers,a list of root vulnerabilities can be found on `http://androidvulnerabilities.org/all`, allows the "hacker" to upload a special program called su to the phone, su provides root access to programs that request it usually superuser permissions bundled with root approve/deny requests from applications who want root replaces convetional password with approve/deny, not secure but much more convenient rooting the phone modifies the software thus can brick the phone, meaning the phone is nonfunctional since the software is broken

rooting has benefits access all files on the phone, even those which the normal user has no permissions for, modify add delete

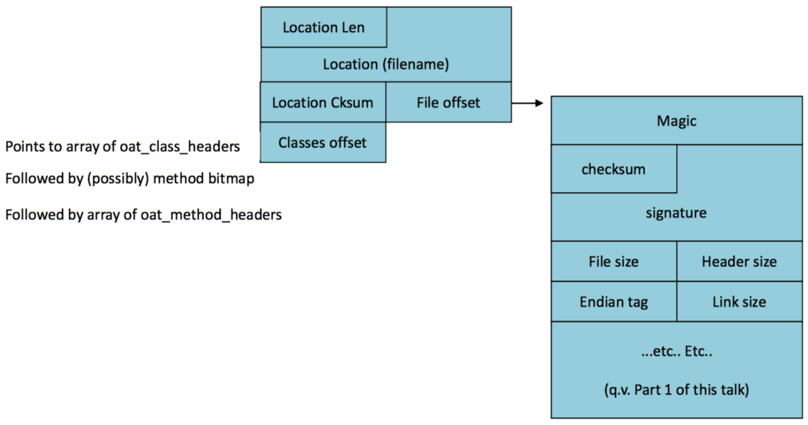


Figure 2.9: oatdex

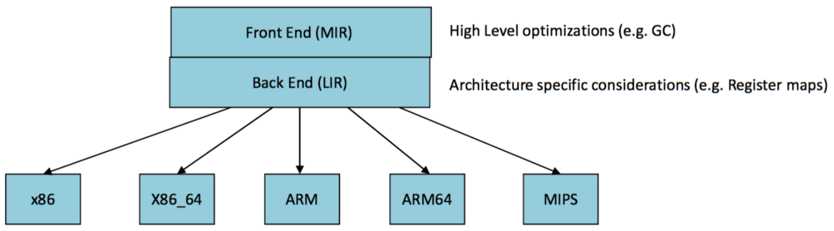

Front End (MIR)
High Level optimizations (e.g. GC)
Back End (LIR)
Architecture specific considerations (e.g. Register maps)
x86
X86_64
ARM
ARM64
MIPS

Figure 2.10: artarch

examples modify system variables, e.g. to utilize the notification led on motorola devices which is usually disabled install custom roms

but there are downsides as well android's content protection are invalid when rooted DRm can be bypassed coupled with dex decompilkation big problem, app can be decompiled, modded and repackaged[33]

2.3 License Verification Libraries

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

Since the original approach of subsection **??** was voided, another method had to be introduced due to ineffectiveness and rising pressure from developer community, google as owner of android and its biggest sore released lvl - `https://developer.android.`

`com/google/play/licensing/overview.html` auch für andere stores interessant da selbe probleme und wollen developer binden amazon (eigenes ökosystem, pushen mit underground und billigen tablets die als einstieg dienen und app store soll generieren), samsung (wollen was besonderes sein, wie apple, spezielle apps für galaxy/note devices, haben auch eigene services), gibt auch chinesische aber die nicht genau betrachtet da nur in china relevanz und westliche eher auf westlichen market (vllt besonderheiten), ausserdem proivder noch [37]

put copy protection methods into app itself, kind of DRM

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/transfering 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

2.3.1 Google's License Verification Library

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

network service, queries Licensing Server from Google check whether current user has license library provided by google transparent since google delivers code

[37]

manages a connection between your app and Google Play, performs a license check with server to see if valid license (.e.g purchased from google play) Digital Rights Management (DRM) [30]

introduced to fight piracy in Google Play, introduced 07/27/2010 simple and free service [16]

Functional Principle

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

integrate into application by developer, allows simple checking and callback process with google asks google play app whether the app was bought on the appstore by the user, takes care of the complicated process (webservice, network etc) on respond google app passes response to the callback

determine license status of an app, licensing service needs two informaiton: - package name of app, a nunce that has to be present in every server response to esnure attacker security, callback for async handling of server respone, implemented in initial license

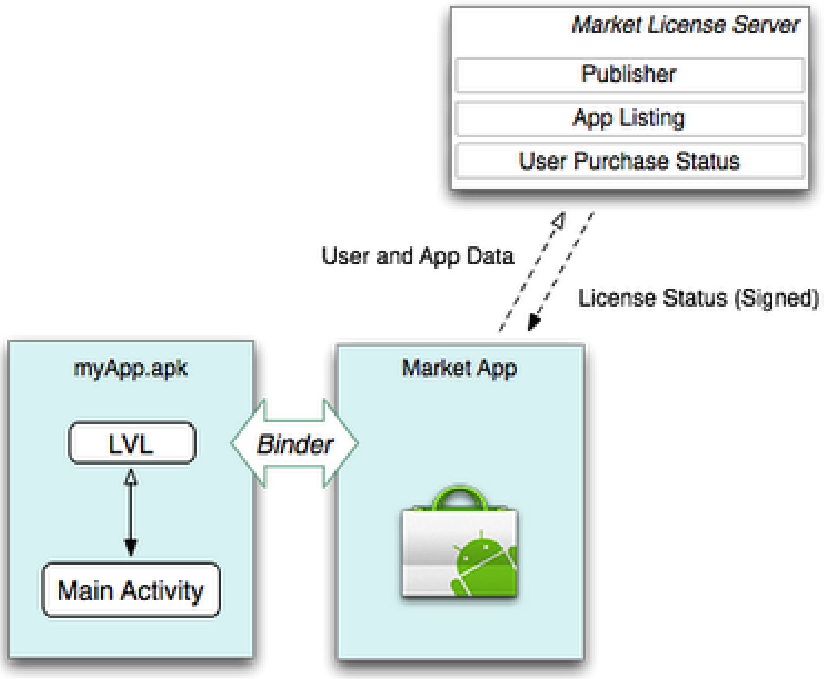


Figure 2.11: lvl [16]

check - user - specific info user and device such as primary google acc and imsi, collected and provided by google play - google play

security of response important every published app in playstore, play generates public/private key pair, developer gets public key public key is embedded into app google play licensing server signes response with app private key public key used to check signature of response, effective mechanism is established to ensure origin and detect tampering [37]

The information about the application, the device and the user goes off to Google's servers. Google then checks your name against the list of people it knows have paid for the application on Google Play. (It could also check the name of the application against a list of applications it knows that you've downloaded from Google Play) If it can see that you have downloaded (and paid for) the application from Google Play it sends back that you have a license, if not then it tells the app you don't.

request starts when app initiates to service on Google Play client application Google Play sends request to licensign server and receives result google play passes it to app and app decides what happens [8]

network based service lets query trusted Google Play licensign server, determine whether application is licensed for current user based on buyers database when, and how often, you want your application to check its license control over how it handles the response, verifies the signed response data, and enforces access controls

need internet, Google account (else you werent able to buy the applciations), google play installed addng licensing to app does not affect way the app functions when run on a device that does not offer Google Play

saves license on device with timed life

replace as copy protection flexible, secure mechanism for controlling access to applciation replaces copy protection mechanism which is no longer supported that was previously offered on Google play license based model that is enforceable on all devices that have acces to google play access is not bound to characteristics of host device, but google play and licensing policy definded can be installed, manages on any device and any storage, even SD card [8]

Implementation of the License Verification Library

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

run time, set of libraries provided by google app can query Google Play licensing Server to determine the license status of user returns information on whether your user is authorized to use the app based on stored sales revords real time over network [16]

what you need

google publisher account on google play google play developer console at Services & API, app specific public key for licensing, implement it into the application copy it later into the app

[6]

- Adding the licensing permission your application's manifest

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

Code Snippet 2.1: Calling the LVL

- Implementing a Policy, provided by LVL or own - Implementing an Obfuscator, if Policy caches any license response data. **??** line 59, cannot be reused or manipulated by a root user - adding code to check license in application main activity_main [6] `https://developer.android.com/google/play/licensing/setting-up.html`

`https://developer.android.com/google/play/licensing/adding-licensing.html`

```
57      final String mAndroidId = Settings.Secure.getString(this.getContentResolver(),
                Settings.Secure.ANDROID_ID);
59      final AESObfuscator mObsfuscator = new AESObfuscator(SALT, getPackageName(),
            mAndroidId);
```

```
        final ServerManagedPolicy serverPolicy = new ServerManagedPolicy(this, mObsfuscator);
        mLicenseCheckerCallback = new MyLicenseCheckerCallback();
        mChecker = new LicenseChecker(this, serverPolicy, BASE64_PUBLIC_KEY);

        mChecker.checkAccess(mLicenseCheckerCallback);
```

Code Snippet 2.2: Calling the LVL

```
        @Override
        public void allow(final int reason) {
            ...
        }

        @Override
        public void dontAllow(final int reason) {
            ...
        }

        @Override
        public void applicationError(final int errorCode) {
            ...
        }
```

Code Snippet 2.3: Callback

2.3.2 Amazon DRM

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

Amazon wants piece of Android pie, also earn money from selling apps, alternaticve to Google play Amazon introduced its appstore on the 03/22/2011 for Android and Fire tablets comes with own DRM which is free to enable/disable by developer[4] since the Google LVL only works with Google Play named Kiwi (taken from decompilation) store is completely independent [5]

Functional Principle

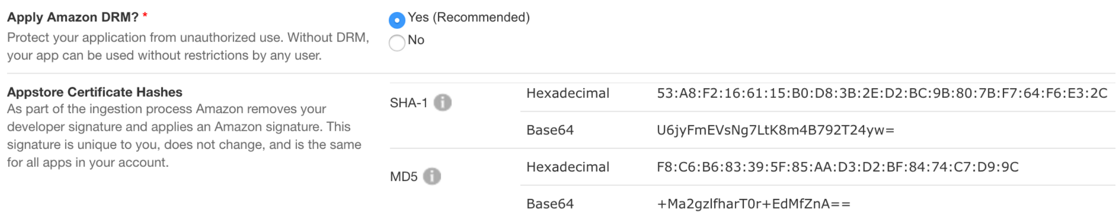
This is my real text! Rest might be copied or not be checked!

sis is text was sind voraussetzungen? amazon app, account active der die app hat

high level prerequisites amazon developer account

when uploading the app, user is asked whether

different approach to perform license verification and enforce result google lvl include and integrate modified version of lvl library, not required to implement any mechanism



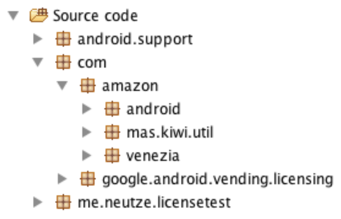
Apply Amazon DRM? *
Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.

[✓ 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	+Ma2gzlfharT0r+EdMfZnA==

Figure 2.12: amazon



```
▼ Source code
    ▶ android.support
    ▼ com
        ▼ amazon
            ▶ android
            ▶ mas.kiwi.util
            ▶ venezia
        ▶ google.android.vending.licensing
    ▶ me.neutze.licensetest
```

Figure 2.13: amazonFolder

on their own, done by amazon packaging tool when submitting can check amazon DRM (see picture), applay amazon DRm to "Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user." as the description says "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." as the description says, so developer signing the application by the develoepr before submitting is not necessary, amazon decompiles apk, injects drm code, compiles it and signs it with the "amazon developer" certificate [37]

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

Implementation of Kiwi

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

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

[37]

```
    public void onCreate(Bundle bundle) {
        onCreateMainActivity(bundle);
        Kiwi.onCreate((Activity) this, true);
    }
```

Code Snippet 2.4: Callback

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

2.3.3 Samsung DRM

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

Samsung as a major player in the smartphone business has also his own app store [17] Galaxy Apps by Samsung, formerly known as Samsung Apps, for devices by Samsung renamed in July 2015 called zirconia for android [43]

Functional Principle

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

library probiding reventive measure against illegal repoduction 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 damkages, 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, infromation of device and applciation will be send 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 **??** does not work if network is offline or in airplane

[43]

Implementation of Zirconia

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

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 basics steps (see **??**) create
can be implemented in any stage of the application, e.g. start or when saving
[43]

```
        final Zirconia zirconia = new Zirconia(this);
        final MyLicenseCheckListener listener = new MyLicenseCheckListener();
        listener.mHandler = mHandler;
        listener.mTextView = mStatusText;
        zirconia.setLicenseCheckListener(listener);
        zirconia.checkLicense(false, false);
```

Code Snippet 2.5: Creating Zirconia

```
        @Override
        public void licenseCheckedAsValid() {
            mHandler.post(new Runnable() {
                public void run() {
                    ...
                }
            });
        }

        @Override
        public void licenseCheckedAsInvalid() {
            mHandler.post(new Runnable() {
                public void run() {
                    ...
                }
            });
        }
```

Code Snippet 2.6: Callback

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 counter measurements 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. [31]

dex to java .dex and .class are isomorphic dex debug items map dex offsets to java line numbers tools like dex2jar can easily decomile from dex to a jar extremly useful for reverse engineering, even more so useful for malice
flow from dex to java is bidirectional, decompile code back to java, remove annoyances

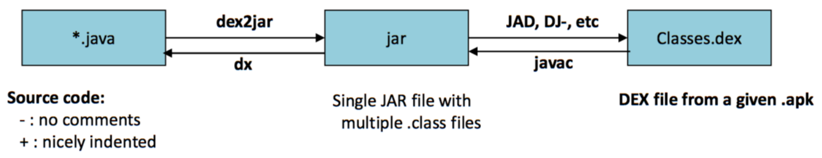


Figure 2.14: re1

like ads, registration, uncover sensitive data (app logic, secrets), replace certain classes with others (malicuous ones), recompile back to jar, then dex, put cloned/trojaned version of your app on play or other market
[34]

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 mgiht 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

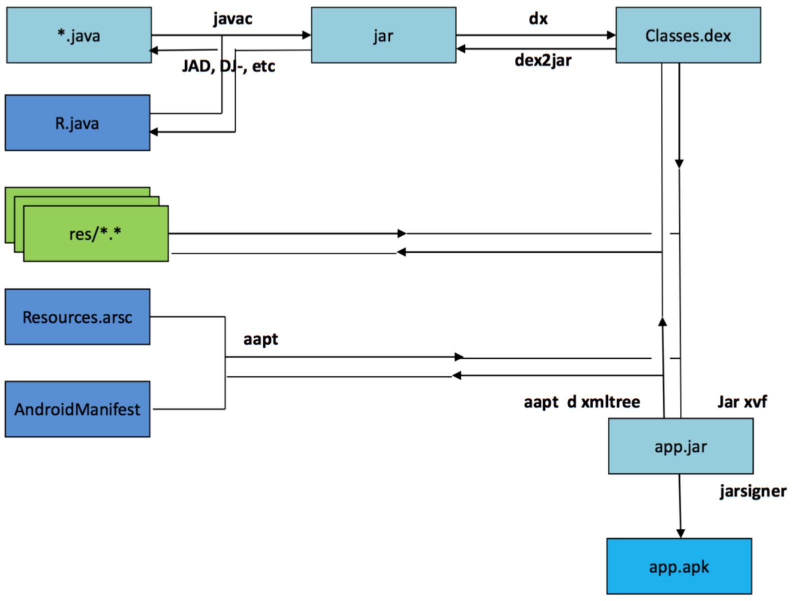


Figure 2.15: re2

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?

`https://mobilesecuritywiki.com/`
`https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf`
main tools

Getting an APK

5 most apps are installed /data/app, android restricts access, with root possible

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

then find app which should be transfered to computer and use adb pull /data/app/me.neutze.licensetest 1/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

[37]

In the following there will be an example application to generalise the procedure. The application is called L̈icenseTeständ 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

Dex

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

[34] 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

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

RESULT OUTPUT

```
a6 8e 15 00 bd 8e 15 00 d5 8e 15 00 f0 8e 15 00 |................|
```

Code Snippet 2.7: Quelle

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)
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

Smali Code

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) [31]

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)
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

Java

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
probleme des disassemlbers erklären
interpretations sache
deswegen zwei compiler
unterschiedliche interpretation resultiert in flow und auch ob sies können ist unterschiedlich

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

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

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 [31]

`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

RESULT OUTPUT code Listing

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)

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

jadx

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

RESULT OUTPUT code Listing

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)

https://github.com/skylot/jadx

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

Es gibt noch mehr tools, wurden angewendet und verglichen, aber diese waren die haupttools und haben ihren dienst erfüllt

Comparison of Code

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 gegebenem 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

https://wiki.ubuntuusers.de/diff

-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 subdirectories.
script erklären

wo findet man es?
welches level?
vorteil
blabla aus dem internet

3 Cracking Android Applications with LuckyPatcher

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

There are plenty of applications which can be used to modify Android apps. This thesis focuses on the on on device cracking application LuckyPatcher, especially on its license verification bypassing mechanism. reason lucky ptcher is taken exremely popular and easy(no technical knowdlege except root but apps can be traded) to use and discusses a lot in android community develoeprs/users because of damage/advantage

3.1 What is LuckyPatcher and what is it used for?

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

main goal is to circumvent license verification, app should behave as it is legally aquired in app store, by user hence work normally, full features

most common way client-server license verification, app gathers info and sends to server, server checks info and depending on this sends response, finally app acts according to response code

since server is not accessible and man-in-the-middle has to break encryption, like spoofing, which is difficult (encryption in general), has to work in application

effective, popular, vielseitig (viel internet -see- quelle) high damage potential since popular, automated and general use by non professional

[37]

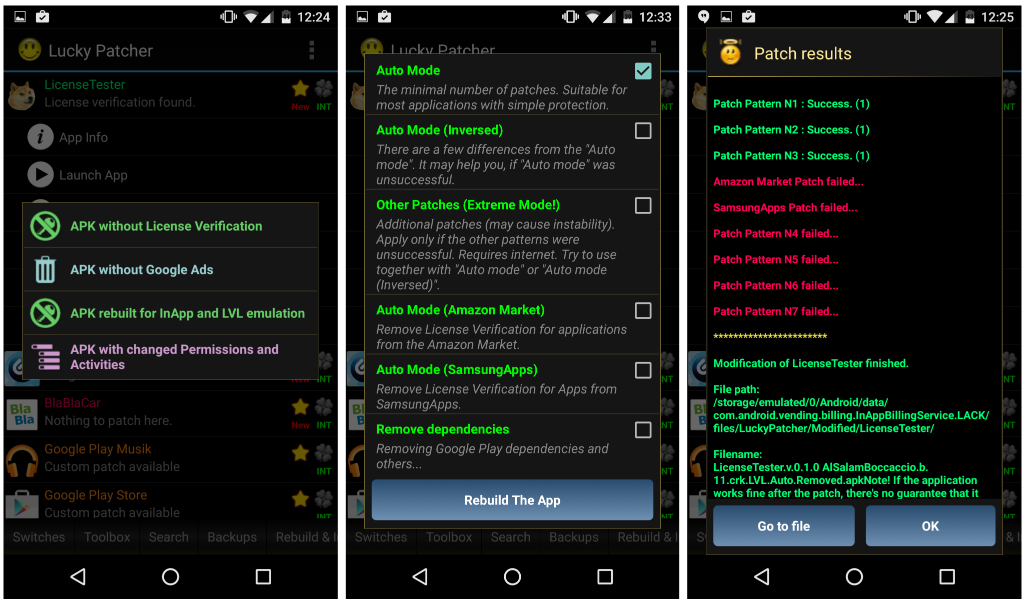
written by ChelpuS

for this master's thesis the version 5.9.5 written bei chelphus requires root and busybox, an application which provides standard UNIX tools for Android[51] apply patches: - Remove license check in premium apps (used to crack DRM) - remove ads -Customize and restrict permissions and activities -Create a modified app (means an APK file to install the app with a patch already applied)

not 100percent warranty that patching works due to modified libraries

erklären wie man ihn gestestet hat, woher die apps, nachgefragt ob ok etc

LuckyPatcher is described as following on the offical webpage: "Lucky Patcher is a great Android tool to remove ads, modify apps permissions, backup and restore apps,



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)
Remove License Verification for applications from the Amazon Market.

Auto Mode (SamsungApps)
Remove License Verification for Apps from SamsungApps.

Remove dependencies
Removing Google Play dependencies and others...

Rebuild The App

```
Patch Pattern N1 : Success. (1)
Patch Pattern N2 : Success. (1)
Patch Pattern N3 : Success. (1)
Amazon Market Patch failed...
SamsungApps Patch failed...
Patch Pattern N4 failed...
Patch Pattern N5 failed...
Patch Pattern N6 failed...
Patch Pattern N7 failed...
************************
Modification of LicenseTester finished.
File path:
/storage/emulated/0/Android/data/
com.android.vending.billing.InAppBillingService.LACK/
files/LuckyPatcher/Modified/LicenseTester/
Filename:
LicenseTester.v.0.1.0 AlSalamBoccaccio.b.
11.crk.LVL.Auto.Removed.apkNote! If the application
works fine after the patch, there's no guarantee that it
```

Figure 3.1: Left: Features offered LuckyPatcher
Middle: Variants to crack license verification
Right: Result after patching

bypass premium applications license verification, and more. To use all features, you need a rooted device." [15]

install apk from playstore -see- have root -see- open lucky -see- chose mode

this thesis focuses on the removing of licensing check, angewendet auf verschiedene apps aus den jeweiligen stores

3.2 Modus Operandi

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

analysis done by looking at patched applications since luckypatcher has done a lot of obfuscation etc which makes it almost impossible to undertand (tools crash, junkbyte injection, only a few giant classes in java which do not make sense, some classes not decompiled since crash, dex/smali not very useful since no names etc), poor decompilation result hard to analyze since of obfuscationa nd anti-decompilation measure

so look at cracked application in order to udnerstand how lucky aoptcher works diff between code of unpatched and patched version of application which have lvl included first it was tested with a simple app which was created and implemented lvl also for apps from app store to see behaviour diff tool to compare code base and see positions lucky patcher attacks

multiple tools used to reverse engineer the license verification circumvention by lucky patcher (see tools from reengineering) first look at java code (higher representation, human readable), to understand what/where class/function have been modified smali for having names for the bytecode executions dex to see exactly how it was modified document intended change by luckypatcher

luckypatcher works by patching dex (patching based not call intercept)

most patching options target googles lvl, but also available for amazon and samsung

information if lvl is contained can be pulled from manifest because of com.android.vending.CHECK_LIC
permission, only to show user and can circumvent anyway since solely for user to flag in market (ist das noch so?), trying to trick and not declare would break violate permissions and be stopped by andorid

luckypatcher extracts original classes.dex from apk and patches it patching is doen on binary level and done by using different patterns which are dependent on the modus (see patterns/modi) the result is either used to substitude the optimized classes.dex of attacked app with the patched .odex in dalvik cache (/data/dalvik-cache/), or can be output as an apk

works on most applications but has few problems on custom modified lvl libraries (liste der apps wo es funzt aus gdrive tabelle)

after applying patterns LP needs to update file header to reflect new checksum and hash values

[37]

wo arbeitet er?
warum dex und nicht odex anschauen?

Since the code is modified directly a static analysis is sufficient.

UM ES EINFACHER ZU MACHEN, KEINE ODEX (WARUM), APK CREATEN UND AUF EINEM NORMALEN HANDY INSTALLIEREN(dann sieht man dass man die app wem anders gecracked geben kann - ringschluss blackmarket)

after modification the dex is signed again in order to work on the phone (see installation und signature)

WIE IST MEIN VORGEHEN? aufgrund dass odex datein device spezifisch sind und allgemeiner ansatz gesucht wird, wird die app playstore -> modified APK erstellen -> analysieren

3.3 Variants for Cracking License Verification

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

patterns und patching modes grob erklären (modi von luckypatcher die verschiedene operationen (pattern) auf app anwenden) => vorgehensweise zur

see figure **??** middle liuckypatcher offeres different sets of methods to remove the license verification Auto Mode - "The monomal number of patches. Suitable for most applications with simple protection" - uses patterns

Auto Mode (Inversed) - "There are a few differences from the ″Auto mode″. It may help you, if "Auto mode" was unsuccessful." - uses patterns

Other Patches (Extreme Mode!) - "Additional patches (mnay cause instability). Apply only if the other patterns were unsuccessful. Requires internet. Try to use together with ″Auto mode″ or ″Auto mode (Inversed)″." - uses patterns

Auto Mode (Amazon Market) - "Removes License Verification for applciations from Amazon Market" - uses patterns

Auto Mode (SamsungApps) - "Removes License Verification for Apps from SamsungApps" - uses patterns (is now GalaxyApps)

in order to find out what patterns are doing, different apps had to be analysed after patching the apps chosen were already owned, in addition an app for each license verification model was created, uploaded and installed from the store so the license verification was working

to verify that license check is enabled, each app was extracted from the device using method described in 2.4 and installed on a device with a different google account then for each app a modified apk see figure **??** left, using one modus is created and copied to a computer for further inspection. so for each app there are 5 modified apks now

as example apps to show results Runtastic Pro[42], Version 6.3, the created LicenseTest and Teamspeak 3[56], Version 3.0.20.2, are chosen

the result after patching the different apps with each modus returned the patterns used by each modus

3.4 Patching Patterns

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

In order to identify the single patterns, the information from the output of cracking **??** right, of the apps was matched with the changes in the code. the changes in the code were inspected on dex, smali and java level with the tools explained in Section 2.4. in case of LVL, from the information where in the package the change was done,

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 3.1: Functionality for the test apps before and after patching

conclusion to the original class from in the lvl could be done

diff for original app and modified app

example code taken from an app which was inspected, modification happens for all at the same spot/manner

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

Pattern N1

classes it attacks `com/google/android/vending/licensing/LicenseValidator` LicenseValidator, responsible for decrypting and verifying the response from the licensing server[19]

```
@@ 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

values are swapped

```
@@ 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

switch case for input 0x01 (not licensed) and 0x02 (old license key) are swapped

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

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

old code when license code not licensed return in case not licensed with error after patching when not licensed return as old license key

Pattern N2
classes it attacks `com/google/android/vending/licensing/LicenseValidator.java` LicenseValidator, responsible for decrypting and verifying the response from the licensing server[19] greift auch zB. google maps api (`com/google/android/gms/`) oder in app billing (`com/android/iab/v3/`) an, collateral schaden one Pattern

```
@@ 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

move-result is replaced by move const

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

Code Snippet 3.5: Diff on Smali level for n2 pattern

instead of moving the result from a function to v3, it is initiated with true/1

```
@@ 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) old code: signature was verified, if true it is continued after patching the verification is treated as always true and so it is continued

Pattern N3
classes it attacks `com/google/android/vending/licensing/APKExpansionPolicy.java` `com/google/android/vending/licensing/ServerManagedPolicy.java` Policy integration of License Verification Library, those are the two examples offered by Google[19]

```
@@ 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

for forward value 0x0 is switched with 0x1 and for inverse...inversed

```
@@ 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

variable is initiated with opposite of what they were initiated before

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

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

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

old code variable is initiated false and true for inversed as basic return value after patching the return is the opposite, meaning all true results are now false and all false are now true, meaning wrong input is declared as OK

Pattern N4
classes it attacks `com/google/android/vending/licensing/LicenseChecker.java` LicenseChecker, class that instatiates a license check[19]

```
@@ 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

if-eqz is repalces by if-ne

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

Code Snippet 3.11: Diff on Smali level for N4 patch

in the original code variable v0 is compared for not equality with zero after it is patched it is always compared with itself which returns always true and the condition is always called

```
@@ Pattern N4 @@
- if(licenseCached()) {...}
+ b = licenseCached();
+ if(b == b) {...}
```

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

in the original code it is checked whether the license is already cached, fi yes, condition is called after patching the result of the check is always compared to itself, and thus the condition is always called

Pattern N5
classes it attacks `com/google/android/vending/licensing/LicenseValidator.java`
works the same way as pattern N2
im gegensatz zu N2 wird jetzt die condition nie aufgeraufen anstatt sie immer aufzurufen wie in N5 result is that the check for the result code given to the function and extracted from the server response is disabled since the result of the check is set to always false

Pattern N6
classes it attacks `com/google/android/vending/licensing/LicenseValidator.java`

```
@@ 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 3.13: Diff on Dex level for N6 patch

if-eqz is replaced by move constant, variables for if-eq are changed

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

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

Code Snippet 3.14: Diff on Smali level for N6 patch

instead of testing for zero and then calling a condition, the to test variable is changed and the condition removed the second equal check is done by comparing a variable with itself thus always true and the conidition is called

```
@@ Pattern N6 @@
- if (p2 == 0 || p2 == v8) {...}
+ p2 = 0;
+ if (v0 == v0) {...}
```

Code Snippet 3.15: Diff on Java level for N6 patch (abstracted)

instead checking two variables for a case, the condition is just always called

Pattern N7
classes it attacks `com/google/android/vending/licensing/ILicenseResultListener.java` ILicenseResultListener, IPC callback implementation, receives async response from server[19] einfach auf alles was in `com/android/` ist, some kind of bruteforce

similar to N2, but Java result is more generic

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

Code Snippet 3.16: Diff on Java level for N7 patch (abstracted)

instead of initializing variable with result from function, it is always initialized with false / 0

Amazon
also applies pattern N2

classes it attacks, inside kiwi logic `com/amazon/android/licensing/b.java` `com/amazon/android/o/d.java`

see, obfuscated

similar like pattern N4

```
@@ 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 3.17: Diff on Dex level for Amazon patch

if-eqz is repalces by if-ne

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

Code Snippet 3.18: Diff on Smali level for Amazon patch

in the original code variable v0 is compared for not equality with zero after it is patched it is always compared with itself which returns always true and the condition is always called

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

Code Snippet 3.19: Diff on Java level for Amazon patch (abstracted)

in the original code the result from the server is tested whether it is "LICENSED" after patching the response is always evalauted and the result is compared with itself which is always true

result never the less what the check for "LICENSED" returns, the condition for "LICENSED" is always called

Samsung Pattern
also applies pattern N2

classes it attacks, inside zirconia logic `com/samsung/zirconia/LicenseRetriever.java` `com/samsung/zirconia/Zirconia.java`

not obfuscated

two patterns, lets call it S1 and S2, S1 used on both, S2 used twice on zirconia

```
@@ 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 3.20: Diff on Dex level for Samsung patch

S1 input for if-eq is modified S2 move-result is replaced by move const

```
@@ 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 3.21: Diff on Smali level for Samsung patch

S1 in the original code checks whether to different variables are equal after patching the check is done with the same variables and thus always returns true

S3 in the original code the result of a function is moved to v0 and returned after patching true/1 is always moved to v0 and returned

```
@@ Pattern S1 @@
- if (v6 == 12) {...}
+ if (v0 == v0) {...}

@@ Pattern S2 @@
- return foo();
+ return 1;
```

Code Snippet 3.22: Diff on Java level for Samsung patch (abstracted) `com/samsung/zirconia/LicenseRetriever.java` always starts condition, even though input is not "12" as supposed to start `com/samsung/zirconia/Zirconia.java` S1 always returns true for checkLicenseFile and checkLicenseFilePhase2, does not check anything which is done normally S2 always starts condition, even though input is not as supposed to start `com/samsung/zirconia/Zirconia.java`

	Patterns							
Modus	N1	n3	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	

Table 3.2: Overview of License Verification Library patching patterns applied by each modus

summarizing what patterns each modus applies Table 3.2

auto: just applies minimum patches, N1 swaps switch cases so not licensed is treated here as old license key in the LicenseValidator, N2 skips the signature verification in the LicenseValidator, N3 inverts the return boolean for the policy checks in the implemented Policy class by initializing with 0/false, and N4 skips, in the only case occured in the test set, the check whether download is allowed and allows it always auto inverse: does the same as auto but initializes the policy check with 1/true instead of false extreme: auto+extreme: applies auto and extreme patches auto inverse+extreme: applies auto and extreme patches

!!! kann man das excel sheet in die datein machen und nicht als appendix, da manche apps ihre ergebnisse nicht öffentlich sehen wollen !!!

3.5 Learnings from LuckyPatcher

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

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 [37]

4 Counter Measurements for Developers

Now that that the functionality of LuckyPatcher is analyzed, it is time to investigate in possible solutions for developers. Counter measurements preventing the cracking app from circumventing the license check mechanism are addressed in four different ways. The first chapter covers functions to discover preconditions in the environment cracking apps use to discover weaknesses or need to be functional. The second chapter uses the aquired knowledge about LuckyPatcher to modify the code resulting in the patching being unsuccessful. In the third chapter presents methods to prevent the reengineering of the developerś application and thus the creation of custom cracks. Further hardware and external measurements are explained in the fourth chapter.
general suggestions by google `http://android-developers.blogspot.de/2010/09/securing-android-lvl-applications.html`

countermeasurements can be applied at different levels, when creating the software, when compiling the code to dex and on the dex file itself

goal is to

amazon/samsung not much to do since from company that is why the following not simple methods target lvl

patching application code is both most wide-spread and most powerful to interfer app logic ease is unfortunate for android developers, need better methods to protect vulnerable/precious code from attacks adding additional layer of security, making it a little ahrder for attackers

CHARAKTERIZED IN DIFFERENT TYPES, tampering protection to detect attack, modifications to enforce additional work in order to crack, methods directly targeting reengineering and additional external features, can be stacked [37]

4.1 Tampering Protection

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

applied when programming

Environment and Integrity Checks, wenn die umgebung falsch ist, kann die app verändert werden. deswegen von vornherein ausschließen, dass die bedingungen dafür gegeben sind. [37]

mechanisms should work for amazon/lvl/samsung –see- beweis! (amazon die signature den die seite vorgibt?)

force close im falle von falschem outcome, entspricht nicht android qualität `http://developer.android.com/distribute/essentials/quality/core.html` aber so wird es dem user klarer dass seine application gecracked ist. harmlosere variante dialog anzeigen oder element nicht laden.

es gibt verschieden punkte um die integrity der application sicherzustellen. dies beinhaltet die umgebung debugg oder rootzugriff, die suche nach feindliche installierte applicationen oder checks nach der rechtmäßigen installation und rechtmäßigen code.

also works for samsung and amazon

in order to remove/disable lvl they have to modify the code unless done precisely can be detected by code [28]

4.1.1 Prevent Debuggability

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

WAS IST DIE IDEE DAHINTER? WIE FUNKTIONIERT ES? WIE WIRD ES IMPLEMENTIERT? WIE SIEHT DAS RESULT AUS (EXAMPLE BILD)

der debug modus kann dem angreifer informationen/logs über die application geben während diese läuft, aus diesen informationen können erkenntnisse über die funktionsweise geben die für einen angriff/modifikation gewonnen werden können. aus diesen informationen können dann patches für software wie lucky patcher entwickelt werden, da man die anzugreifenden stellen bereits kennt. kann erzwungen werden indem man das debug flag setzt (wo ist es, wie kann es gesetzt werden)
um dies zu verhindern kann gecheckt werden ob dieses flag forciert wird und gegebenenfalls das laufen der application unterbinden

```java
public static boolean isDebuggable(Context context) {
    boolean debuggable = (0 != (context.getApplicationInfo().flags & ApplicationInfo.
          FLAG_DEBUGGABLE));

    if (debuggable) {
        android.os.Process.killProcess(android.os.Process.myPid());
    }

    return debuggable;
}
```

Code Snippet 4.1: asd[15]

Code SNippet /refCode Snippet: luckycode zeigt eine funktion die auf den debug modus prüft. Dazu werden zuerst in zeile 15 die appinfo auf das debug flag überprüft. ist dieses vorhanden, ist die variable debuggable true. in diesem fall wird dann die geschlossen

4.1.2 Root Detection

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

WAS IST DIE IDEE DAHINTER? WIE FUNKTIONIERT ES? WIE WIRD ES IMPLEMENTIERT? WIE SIEHT DAS RESULT AUS (EXAMPLE BILD)

`http://stackoverflow.com/questions/10585961/way-to-protect-from-lucky-patcher-play-lice`

```java
    public static boolean findBinary(Context context, final String binaryName) {
        boolean result = false;
        String[] places = {
                "/sbin/",
                "/system/bin/",
                "/system/xbin/",
                "/data/local/xbin/",
                "/data/local/bin/",
                "/system/sd/xbin/",
                "/system/bin/failsafe/",
                "/data/local/"
        };

        for (final String where : places) {
            if (new File(where + binaryName).exists()) {
                result = true;
                android.os.Process.killProcess(android.os.Process.myPid());
            }
        }

        return result;
    }
```

Code Snippet 4.2: Partial Listing

SafetyNet provides services for analyzing the configuration of a particular device, to make sure that apps function properly on a particular device and that users have a great experience. `https://developer.android.com/training/safetynet/index.html`
Checking device compatibility with safetynet
Unlocked bootloader doesn't matter. Can't have root installed initially. Has to

be a stock / signed ROM. https://www.reddit.com/r/Android/comments/3kly2z/checking_device_compatibility_with_safetynet/

4.1.3 LuckyPatcher Detection

This is my real text! Rest might be copied or not be checked!
WAS IST DIE IDEE DAHINTER? WIE FUNKTIONIERT ES? WIE WIRD ES IMPLEMENTIERT? WIE SIEHT DAS RESULT AUS (EXAMPLE BILD)

As the example shows, this check is not only a solution to prevent the application from running when LuckyPatcher is present on the device. The screening can be expanded to check for the installation of any other application, like black market apps or other cracking tools as the code example Code Example 4.5 shows.
http://stackoverflow.com/questions/13445598/lucky-patcher-how-can-i-protect-from-it
http://android-onex.blogspot.de/2015/07/anti-piracy-software-activated-solved.html

```
    public static boolean checkInstall(final Context context) {
        boolean result = false;
        String[] luckypatcher = new String[]{
                // Lucky patcher
                "com.dimonvideo.luckypatcher",
                // Another lucky patcher
                "com.chelpus.lackypatch",
                // Black Mart alpha
                "com.blackmartalpha",
                // Black Mart
                "org.blackmart.market",
                // Lucky patcher 5.6.8
                "com.android.vending.billing.InAppBillingService.LUCK",
                // Freedom
                "cc.madkite.freedom",
                // All-in-one Downloader
                "com.allinone.free",
                // Get Apk Market
                "com.repodroid.app",
                // CreeHack
                "org.creeplays.hack",
                // Game Hacker
                "com.baseappfull.fwd"
        };

        for (String string : luckypatcher) {
            if(checkInstallerName(context, string)){
                result = true;
            }

            if (result) {
                android.os.Process.killProcess(android.os.Process.myPid());
            }
        }

        return result;
    }

    private static boolean checkInstallerName(Context context, String string) {
        PackageInfo info;
        boolean result = false;

        try {
            info = context.getPackageManager().getPackageInfo(string, 0);

            if (info != null) {
                android.os.Process.killProcess(android.os.Process.myPid());
                result = true;
            }

        } catch (final PackageManager.NameNotFoundException ignored) {
        }

        if (result) {
            android.os.Process.killProcess(android.os.Process.myPid());
        }
        return result;
    }
}
```

Code Snippet 4.3: Partial Listing

4.1.4 Sideload Detection

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

WAS IST DIE IDEE DAHINTER? WIE FUNKTIONIERT ES? WIE WIRD ES IMPLEMENTIERT? WIE SIEHT DAS RESULT AUS (EXAMPLE BILD)

`http://stackoverflow.com/questions/10809438/how-to-know-an-application-is-installed-fro`
enforces installation from trusted sources to ensure original application developer should only distribute his software on in the app registered soruces, else app wont work an user is annoyed copied apps do not work now

```java
public class Sideload {
    private static final String PLAYSTORE_ID = "com.android.vending";
    private static final String AMAZON_ID = "com.amazon.venezia";
    private static final String SAMSUNG_ID = "com.sec.android.app.samsungapps";

    public static boolean verifyInstaller(final Context context) {
        boolean result = false;
        final String installer = context.getPackageManager().getInstallerPackageName(context.
            getPackageName());

        if (installer != null) {
            if (installer.startsWith(PLAYSTORE_ID)) {
                result = true;
            }
            if (installer.startsWith(AMAZON_ID)) {
                result = true;
            }
            if (installer.startsWith(SAMSUNG_ID)) {
                result = true;
            }
        }
        if(!result){
            android.os.Process.killProcess(android.os.Process.myPid());
        }

        return result;
    }
```

Code Snippet 4.4: Partial Listing

4.1.5 Signature

This is my real text! Rest might be copied or not be checked!
`http://developer.android.com/tools/publishing/app-signing.html`
`http://forum.xda-developers.com/showthread.php?t=2279813&page=5`

CONTRA

The unfortunate side effect of Lucky Patcher working with the Dalvik cache of an app is that the app developers cannot detect manipulations to their code through fingerprinting because the original code, located in "/data/app/<appName.apk>/classes.dex", remains untouched. While it is allowed for an app to access its own optimized byte-code from the cache [32], computing a checksum or a hash for it doesn't make sense because many optimizations to this bytecode are device-specific and cannot be known in advance. [37] !!!überprüfen!!!

Local Signature Check

This is my real text! Rest might be copied or not be checked!
local check whether signature is allowed
once in code
save to use signature in code?

```java
public static boolean checkAppSignature(final Context context) {
    //Signature used to sign the application
    static final String mySignature = "...";
    boolean result = false;

    try {
        final PackageInfo packageInfo = context.getPackageManager().getPackageInfo(context.
            getPackageName(), PackageManager.GET_SIGNATURES);

        for (final Signature signature : packageInfo.signatures) {
            final String currentSignature = signature.toCharsString();
            if (mySignature.equals(currentSignature)) {
                result = true;
            }
        }
    } catch (final Exception e) {
        android.os.Process.killProcess(android.os.Process.myPid());
    }

    if (!result) {
        android.os.Process.killProcess(android.os.Process.myPid());
    }

    return result;
}
```

Code Snippet 4.5: Partial Listing

—> was passiert wenn odex?

4.1.6 Flow Control

This is my real text! Rest might be copied or not be checked!
zweimal LVL und eins failed immer, wenn stumpf modifiziert wird werden beide immer strue und somit erkennt man ob gepatcht wurde

visuelle elemente block für block freischalten, weg der definiert ist, wenn lvl licensed, wenn irgendwo geskippt wird fehlt ein element activate/kill in defined blocks, e.g. if vor switch, noch radikaler

4.2 Library Modifications

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

applied when programming

way to challenge luckypatcher is to actively go against luckypatcher patterns, achieved by modifying the library, see patterns to fight in patterns chapter can be done in different ways, modify library or go native

many developers do not customize the library, easy to hack [37]

goal is to make lvl implementation unique, difficult to trace when decompiled counter intuitive from traditional software engineering viewpoint, removing functions, hiding license check routines in unrelated code

Google is aware of easy hacking and thus suggests modifcations to the lvl modify license verification library in way that it is difficult for attacker to modify the disassembled code and get a positive check as result advantages, harder to crack, cannot be used as blueprint and no blueprint can be used on it, unique [28] ERWÄHNEN DASS IM PROGRAMMIER PROZESS IMPLEMENTIERT

4.2.1 Modify the Library

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

most developers include the LVL unchanged without modifying it, attacks can always be done the same wayso this is very easy for luckypatcher for own security and the security of other developers try to create a unique implementation, ideal would be that no patterns can be applied to you and that in case the implementation gets racked no pattern of yours can be applied to others

this is a list of ideas of how to modify versus each pattern,

pattern 1,7 attacks the switch, the idea is to replace the switch with an if statemant or shuffle the cases move it to a function and implement it somewhere as well thus the code might no longer be together with the rest of the class and the attacker has to specific search for it

pattern 2,4,5 skips using the outcome of a function and setting it always to true, this is a bit harder since it is already in a function, fitting bulk code around can make it harder to detect, especially for patterns, also checking inside again can help with detecting whether the fucntion is tampered, in case it is tampered the app can be killed or elements could not be loaded

pattern 3 modifies the return values on initialization, idea to alter the return value, changing it to e.g. int, thus all the ifs have to be modified to fit the needs

general ideas move the lvl into own application folder replace functions with inline code when possible test again inside the function and e.g. kill the app on tampering best would be to really make it different, e.g. recreate classes

4.2.2 Native Implementation

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

luckypatcher automatic patching works on dex, why not remove the key license verification code from from java and move it to native code, this way luckypatcher has to create a custom recipe for your application since custom recipes can inject .so files

native code can be executed using two thinks, android ndk a toolset to implement native-code languages, e.g. c or c++, and JNI used by java to execute native code

native code is difficult to be decompiled, can be checked for tampering when loading it

4.3 Reverse Engineering Prevention

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

now that the environment is enforced and the lvl is modified, the next goal is to prevent pirates from even starting to analyze the applcation

does not help when standard version is implemented, that is why this is working best with customized implementation of LVL Reverse engineering and code protection are processes which are opposing each other, neither classified as good nor bad
"good" developer: malware detection and IP protection
"bad" developer: analysis for attack and analysis resistance

[31] it is not possible to 100 percent evade reengineering, but adding different methods to hide from plain sight of reengineering tools reengineering cannot be vermiede best is to apply technqiues to make it as hard a possible [37]

if they do not see what the app is doing, they cannot fix it

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.

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

4.3.1 Break Common Reengineering Tools

applied on dex file

One way to prevent reengineering is to break the tools used to reengineer the software can be done by different ways APKfuscator[52] is an example which includes some variants of how to break these tools it was presented on Blackhat 2012 at the talk "dex education: practicing safe"[54] [55] it is a generic obfuscator and munger which works on dex fiels directly simple ideas the propotype has following features

Too long filenames
pirates want to have java to better understanding of code, try to fight there files inside jar have no character limit for names change name of class to alrge but valid name `https://youtu.be/Rv8DfXNYnOI?t=985` works except for the class
breaks only baksmali

Inject bad op code
viele schlagen vor: junkbyte injection well known technique in x86, confuse disassemblers in a way that they produce disassembly errors and disallow correct interpretations, inserting junkbytes in selected locations within the bytecode where a disassembler expects an instruction, junkbyte must take the disassembling strategy into account in order to reach a maximum of obfuscated code, break the two disassebly stragien from 2.4.0, condition for the location is that the junkbyte must not be executed, because an execution would result in an undesired behavior of the application, junkbyte must be located in a basic block which will never be executed [44]

use bad opcode in deadcode
e.g. reference not inited strings [55]

Abuse differences between Dalvik and Java
include code that is legal in dalvik world but not in java world e.g. recursive try/catch as described in [55], valid dex code but (might be) impossible in Java code, has to be

implemented into the class which should not be recovered by the attacker

Increase header size
expected to be 112 bytes, just increase header size which might unexpected to some programms you have to edit every other offset as well

4.3.2 Obfuscation

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

obfuscators are applied when compiling

(a) at source code and (b) bytecode level, Most existing open-source and commercial tools work on source code level
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
[31]

a few dex obfuscators exist, with different approaches 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) can add dead code and dummy lopps (minor impact of performance) can also use goto into other onstructions

[34]

layout obfuscation most programmers name their variables, methods and calss in meaning ful way are preserved in generation of bytecode for dvm, hence still in dex, can be extracted by attacker, gain information and benefit when reengineering mangles names and ifentifiers that original meaning is lost while preserving correctness of syntax and semantics result is bytecode can be interpreted but dissable and decompiule provide meaningless name for identifiers etc, e.g single letters or short combinations, welcome for strings section make it smaller only complicates but does not stop

[37]

will not protect against autoamted attack, does not alter flow of program makes more difficult for attackers to write initial attack removing symbols that would quickly reveal original structure number of commercial and open-source obfuscators available for Java that will work with Android [28]

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.

hilft nicht direkt, aber um reengineering besser zu machen does not protect directly versus luckypatcher but in case of an custom implementation it makes the process of analyzing the app more time consuming

definition obfuscation, was macht es, wie funktioniert es, wer hat es erfunden, wie wendet man es an

"hard to reverse engineer" but without changing the behavior of this application, was heißt hard to reverse

parallele zu disassembler ziehen

Obfuscation cannot prevent reverse engineering but can make it harder and more time consuming. We will discuss which obfuscation and code protection methods are applicable under Android and show limitations of current reverse engineering tools

The following optimizers/obfuscators are common tools. (dadrin dann verbreitung preis etc erklären)

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 [31]

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

[37]

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 obfuscationa, 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
[44]

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[3]

optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code merges identical code blocks performs optimiztations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [22] [58]

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 encryprion, encrypting app resources, tamper detection, removing logging code [31]

son of proguard, does everything that proguard does its a optimizer and shrinekr, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), applciation tamper protection, removes debug information may increase dex size, memory size; decrease speed [58]

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 [25]

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 [58]

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

[1] [2]

4.3.3 Packers

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

goal of packers it to protect the code from modifications or attacks, good or bad nature, often used by malware as well[60]

launch stub application which unpacks and/or decrypts zipped code works at runtime, the 'outer' stub application unpacks content of 'inner' executable into memory and executes, example for x86 platform, UPX [36], since works at runtime hidden from static analysis

dynamic code loading best would be an application is transformed by obfuscator that it does not contain any meta information or directly interpretable bytecode, not possible because DVM would not be able to read -see- packing, often used in malware [26]
packer transforms code that a reverse engineer cannot directly extract information, e.g. by encrypting programm data no information can be extracted without decrypting, would be bad for programm as well that is why packer uses loader stub to decrypt in runtime, soultion decrypt by hand or dynamic analysis
in general two components are created, loader stub and encrypted app, on android the encrypted dex file, easier to create as the loader stub
BILD WIE ER FUNKTIONIERT step 1 load encrypted app into memory (download from server, extract from data structure, plain file available) step 2 app file is decrypted -see- original dex, can be any encryption from simple to hard, speed may slow down step 3 load decrypted dex into DVM from a bytearray, see [44] step 4 execute
[40][61]

[44] stellt hier basic version vor, bessere versionen ist das im folgenden

result: protection makes it hard to analyze the target application, because its bytecode is only available encrypted, decrypted version the unpacking stub has to be analyzed, great slow down, other obfuscations can be applied on stub

program delivered to end user should be transformed, no direct link between it and original source code, attackers cannot reengineer code and find vulnerability

[37]

HoseDex2Jar
HoseDex2Jar is a prototype/proof of concept of a service which obfuscates, mangles and mungs APK files goal it to prevent easy decompilation and thus analysis

original dex file is packed into a compressed container, encrypted and injects into the dalvik header, dexception[55], easy to spot since dex header is not normal static analysis will not work since encrypted file will be ignored, which contents the actual code

on execution loader stub decrypts the dex file in memory and dumps to file system loader stub acts as proxy and passes events to the dex file on system using a dexClass-Loader since the stub is added slight file size increase, but no big impact not available for real use, website taken down further developed projects are bangcle or IJIAMI, often for malware[60][61]

[38][53][58]

ApkProtect
also known as dexcrypt, appears to be active but no website, there might be clones slightly different than a packer, stub fixes broken code which is normally not translated by tools and makes it as it should be features anti-debug, anti-decompile, almost like a packer, string encryption

tool mangles code original code -modifies entrypoint to loader stub -prevents static analysis on intall dalvik optimizes the dex file into memory ignoring bad parts

when executed the 'original' dalvik code is executed, during runtime loader stub from the 'original' code fixes the odex in memory, then it is executed as normal results in slight file size increase and prevents easily static analysis

still has string encryption (like DexGuard, Allatori) afterwards [58][40][61]

4.4 External Support

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

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

4.4.1 Service-managed Accounts

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

By introducing service managed account, the user has to login on a server in order for the application to work an example is spotify, where the user logs in on a server and then can stream music without account it is not possible to obtain the stream details and even though when the stream details are obtained they can be encrypted with a user specific key the attacker does not have this way also an awesome algorithm can be moved onto the server that the application is only a thin client, this way reverse engineering or pirating does not offer any value to the attacker

4.4.2 ART endlich durchsetzen

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 counter measurements against it

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

4.4.3 Secure Elements

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

can either be mounted in the sdcard slot or using an adapter for the usb interface accessed over reads and writes to the filesystem since it has to be small as a sd card and powered by the host system its hardware capabilities are restrained as well [46] with power as low as 25MHz complexe calculations would take too much time

can be used to prevent static analysis encrypted strings from the application can be decrypted by the secure element store property settings on it

no complexe tasks since low power

`https://www.youtube.com/watch?v=rSH6dnUTDZo`

4.4.4 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?

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 priviliged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitve processing offloaded to protect information from malware

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

example Trustzone, Knox

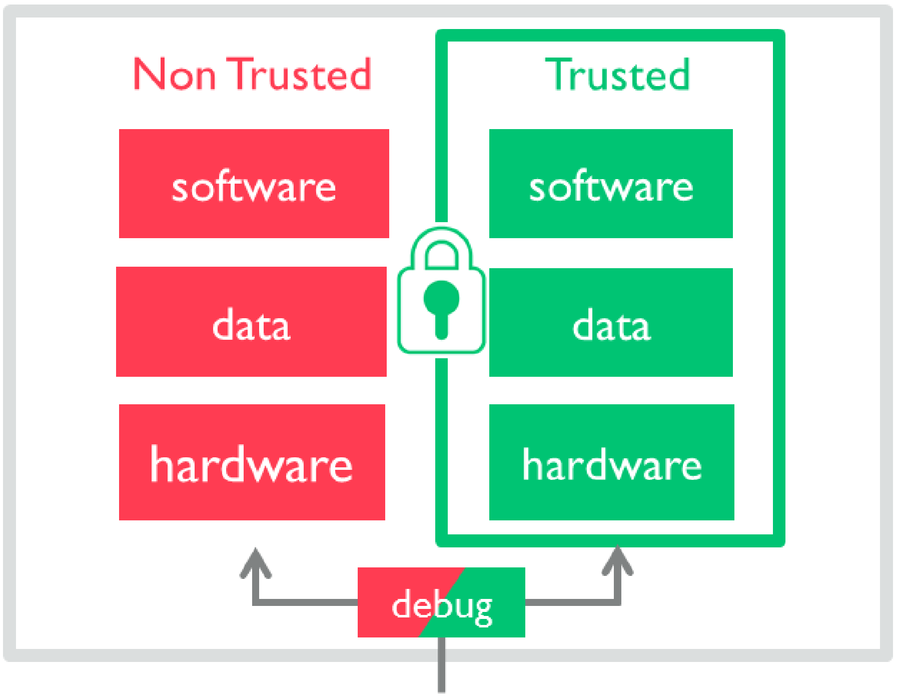


Figure 4.1: tee [10]

[14][10]

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

5 Evaluation of Counter Measurements

Now that the counter measurements are presented, it is time to evaluate their potential versus LuckyPatcher. But not only their capabilities are important for developers but their practicability as well, e.g. price or effort to implement. Each measurement will be looked at in the following.

5.1 Tampering Protection

all tampering counter measurements have kind of the same pattern, boolean check, simple method == simple fix, can be nulled easily when code is known, just as easy to crack as LVL when you know the code, but attackr has to invest some time to understand code and to build counter measurement, in addition with Section 5.3 this can get annoying, evtl create native versions because harder to crack even though it is simple it adds a little bit extra work to attack and when cobined this grows exponential

but be careful because annoy people who want to use root annoy people who bought the app but have luckypatcher/root as well

!!!signature problem mit maps überprüfen!!!

5.2 Library Modifications

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

this should be the first solution anyone does in order to actively fight luckypatcher

5.2.1 Modify the Library

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

this tries to directly encounter luckypatcher by fighting the way the patterns work

it is easy to apply and there are no limits of variants how it can be modified luckypatcher has to look at each app individual and in the worst case create custom patches but as it seams developers do not like to modify as seen in analysis since all apps are somehow patchable

5.2.2 Native Implementation

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

at first a good protection against luckypatcher since the automatic patching discussed in Section **??** does not work for the app any longer, but not protection of luckypatcher because it supports custom patching when analyzing luckypatcher there can be .so files be found which are used to replace existing ones in order to patch fucntionality

so now attackers have to invest time in analyzing the native code difficult since in opposite of byte-code, it does not contain a lot of meta-data such as local variable types, class strcture, which allows bytecode to be compiled on multiple devices, this information is discarded in native code in the compilation process

there are two scenarios: first scenario developers creates his own version of native implementation unique, but developer has to create it itself, knowledge and skill is needed and time has to be spent if done right, safe and advantage that the attacker has to invest time for this app itself in order to analyze the native code, then find a method to break it and repack it and make it available as custom patch, scares off attackers since a lot of work, have to evaluate whether app is worth it so if the resources are available, best method

second scenario one public native library provided by google but when all use the same library, it is a vulnerable point because it makes sense for attackers to analyze the library and try to come up with a patch, since one solution can be applied to many different apps which would justify a lot of work as described in the first scenario, this patch then can then be applied as a custom patch via luckypatcher

for this reason when coming up with a native implementation for a library for all, two things should be included - users do not custimize library so a heavy obfuscation should be applied - since there is only one version, make it as hard to reengineer and predictable as possible, use encryption and dynamic code generation, automatically custimize itself for every app and ever time its loaded this needs a lot of research, work and if it would be so simple, big companies like google would already have come up with it, but in this direction it has to go when stick around for a long time with dex and its vulnerabilities

reengineering kann aushebeln

NATIVE Als ein eigenstaendiges Kapitel koenntest du auch noch untersuchen, wie sich Java-Code und Native-Code am Besten mit einander verflechten lassen, um optimalen Schutz gegen den Lucky Patcher zu gewaehrleisten.

Erste Ideen gab es dazu ja bereits - auch von anderen, wie etwa die Verschluesslung von Inhalten und Dekodierung im native Code unter Verwendung von Secure Elements oder Manipulation von Speicherwerten ueber native Libraries, sodass man die Aenderung im (Java) Smali-Code gar nicht mitbekommt. Letzteres hat Herr Hugenroth in einer Seminararbeit einmal grob skizziert (liegt Dir das vor?). Vielleicht fallen Dir weitere Optionen ein? Auch theoretische Ideen sind willkommen.

5.3 Reverse Engineering Prevention

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

now that the environment is enforced and the lvl is modified, the next goal is to prevent pirates from even starting to analyze the applcation

does not help when standard version is implemented, that is why this is working best with customized implementation of LVL Reverse engineering and code protection are processes which are opposing each other, neither classified as good nor bad
"good" developer: malware detection and IP protection
"bad" developer: analysis for attack and analysis resistance

[31] it is not possible to 100 percent evade reengineering, but adding different methods to hide from plain sight of reengineering tools reengineering cannot be vermiede best is to apply technqiues to make it as hard a possible [37]

if they do not see what the app is doing, they cannot fix it

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.

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

5.3.1 Break Common Reengineering Tools

applied on dex file

One way to prevent reengineering is to break the tools used to reengineer the software can be done by different ways APKfuscator[52] is an example which includes some variants of how to break these tools it was presented on Blackhat 2012 at the talk "dex education: practicing safe"[54] [55] it is a generic obfuscator and munger which works on dex fiels directly simple ideas the propotype has following features

Too long filenames
pirates want to have java to better understanding of code, try to fight there files inside jar have no character limit for names change name of class to alrge but valid name `https://youtu.be/Rv8DfXNYnOI?t=985` works except for the class
breaks only baksmali

Inject bad op code
viele schlagen vor: junkbyte injection well known technique in x86, confuse disassemblers in a way that they produce disassembly errors and disallow correct interpretations, inserting junkbytes in selected locations within the bytecode where a disassembler expects an instruction, junkbyte must take the disassembling strategy into account in order to reach a maximum of obfuscated code, break the two disassebly stragien from 2.4.0, condition for the location is that the junkbyte must not be executed, because an execution would result in an undesired behavior of the application, junkbyte must be located in a basic block which will never be executed [44]

use bad opcode in deadcode
e.g. reference not inited strings [55]

Abuse differences between Dalvik and Java
include code that is legal in dalvik world but not in java world e.g. recursive try/catch as described in [55], valid dex code but (might be) impossible in Java code, has to be implemented into the class which should not be recovered by the attacker

Increase header size
expected to be 112 bytes, just increase header size which might unexpected to some programms you have to edit every other offset as well

5.3.2 Obfuscation

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

obfuscators are applied when compiling

(a) at source code and (b) bytecode level, Most existing open-source and commercial tools work on source code level
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
[31]

a few dex obfuscators exist, with different approaches 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) can add dead code and dummy lopps (minor impact of performance) can also use goto into other onstructions

[34]

layout obfuscation most programmers name their variables, methods and calss in meaning ful way are preserved in generation of bytecode for dvm, hence still in dex, can be extracted by attacker, gain information and benefit when reengineering mangles names and ifentifiers that original meaning is lost while preserving correctness of syntax and semantics result is bytecode can be interpreted but dissable and decompiule provide meaningless name for identifiers etc, e.g single letters or short combinations, welcome for strings section make it smaller only complicates but does not stop

[37]

will not protect against autoamted attack, does not alter flow of program makes more difficult for attackers to write initial attack removing symbols that would quickly reveal original structure number of commercial and open-source obfuscators available for Java that will work with Android [28]

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.

hilft nicht direkt, aber um reengineering besser zu machen does not protect directly versus luckypatcher but in case of an custom implementation it makes the process of analyzing the app more time consuming

definition obfuscation, was macht es, wie funktioniert es, wer hat es erfunden, wie wendet man es an

"hard to reverse engineer" but without changing the behavior of this application, was heißt hard to reverse

parallele zu disassembler ziehen

Obfuscation cannot prevent reverse engineering but can make it harder and more time consuming. We will discuss which obfuscation and code protection methods are applicable under Android and show limitations of current reverse engineering tools

The following optimizers/obfuscators are common tools. (dadrin dann verbreitung preis etc erklären)

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 [31]

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

[37]

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 obfuscationa, 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
[44]

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[3]

optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code merges identical code blocks performs optimiztations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [22] [58]

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 encryprion, encrypting app resources, tamper detection, removing logging code [31]

son of proguard, does everything that proguard does its a optimizer and shrinekr, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), applciation tamper protection, removes debug information may increase dex size, memory size; decrease speed [58]

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 [25]

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 [58]

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

[1] [2]

5.3.3 Packers

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

goal of packers it to protect the code from modifications or attacks, good or bad nature, often used by malware as well[60] launch stub application which unpacks and/or decrypts zipped code works at runtime, the 'outer' stub application unpacks content of 'inner' executable into memory and executes, example for x86 platform, UPX [36], since works at runtime hidden from static analysis

dynamic code loading best would be an application is transformed by obfuscator that it does not contain any meta information or directly interpretable bytecode, not possible because DVM would not be able to read -see- packing, often used in malware [26]
packer transforms code that a reverse engineer cannot directly extract information, e.g. by encrypting programm data no information can be extracted without decrypting, would be bad for programm as well that is why packer uses loader stub to decrypt in runtime, soultion decrypt by hand or dynamic analysis
in general two components are created, loader stub and encrypted app, on android the encrypted dex file, easier to create as the loader stub
BILD WIE ER FUNKTIONIERT step 1 load encrypted app into memory (download from server, extract from data structure, plain file available) step 2 app file is decrypted -see- original dex, can be any encryption from simple to hard, speed may slow down step 3 load decrypted dex into DVM from a bytearray, see [44] step 4 execute [40][61]

[44] stellt hier basic version vor, bessere versionen ist das im folgenden

result: protection makes it hard to analyze the target application, because its bytecode is only available encrypted, decrypted version the unpacking stub has to be analyzed, great slow down, other obfuscations can be applied on stub

program delivered to end user should be transformed, no direct link between it and original source code, attackers cannot reengineer code and find vulnerability

[37]

HoseDex2Jar
HoseDex2Jar is a prototype/proof of concept of a service which obfuscates, mangles and mungs APK files goal it to prevent easy decompilation and thus analysis

original dex file is packed into a compressed container, encrypted and injects into the dalvik header, dexception[55], easy to spot since dex header is not normal static analysis will not work since encrypted file will be ignored, which contents the actual code

on execution loader stub decrypts the dex file in memory and dumps to file system loader stub acts as proxy and passes events to the dex file on system using a dexClass-Loader since the stub is added slight file size increase, but no big impact not available for real use, website taken down further developed projects are bangcle or IJIAMI, often for malware[60][61]

[38][53][58]

ApkProtect
also known as dexcrypt, appears to be active but no website, there might be clones slightly different than a packer, stub fixes broken code which is normally not translated by tools and makes it as it should be features anti-debug, anti-decompile, almost like a packer, string encryption

tool mangles code original code -modifies entrypoint to loader stub -prevents static analysis on intall dalvik optimizes the dex file into memory ignoring bad parts

when executed the 'original' dalvik code is executed, during runtime loader stub from the 'original' code fixes the odex in memory, then it is executed as normal results in slight file size increase and prevents easily static analysis

still has string encryption (like DexGuard, Allatori) afterwards [58][40][61]

5.3.4 Packers

already cracked `https://www.google.de/search?q=hosedex2jar&oq=hosedex2jar&aqs=chrome..69i57j69i60j69i59j69i60l3.1680j0j7&sourceid=chrome&es_sm=91&ie=UTF-8` `http://www.hotforsecurity.com/blog/mobile-app-development-company-fights-off-android-malwa` `html`

BEISPIELBILDER!!

5.4 External Support

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

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

5.4.1 Service-managed Accounts

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

By introducing service managed account, the user has to login on a server in order for the application to work an example is spotify, where the user logs in on a server and then can stream music without account it is not possible to obtain the stream details and even though when the stream details are obtained they can be encrypted with a user specific key the attacker does not have this way also an awesome algorithm can be moved onto the server that the application is only a thin client, this way reverse engineering or pirating does not offer any value to the attacker

5.4.2 ART endlich durchsetzen

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 counter measurements against it

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

5.4.3 Secure Elements

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

can either be mounted in the sdcard slot or using an adapter for the usb interface accessed over reads and writes to the filesystem since it has to be small as a sd card and powered by the host system its hardware capabilities are restrained as well [46] with power as low as 25MHz complexe calculations would take too much time

can be used to prevent static analysis encrypted strings from the application can be decrypted by the secure element store property settings on it

no complexe tasks since low power

`https://www.youtube.com/watch?v=rSH6dnUTDZo`

5.4.4 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?

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 priviliged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitve processing offloaded to protect information from malware

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

example Trustzone, Knox

[14][10]

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

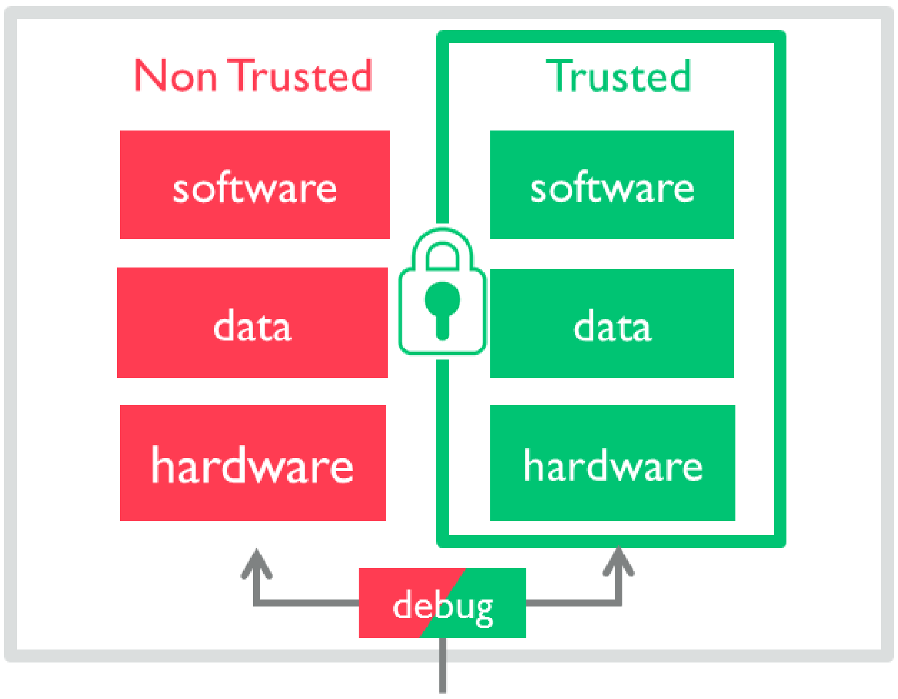


Figure 5.1: tee [10]

6 Conclusion

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

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.
[44]

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 proteciton 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 [37]

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

6.1 Summary

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

jedes chapter beschreiben

6.2 Discussion

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

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, jsut 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 [37]

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 [30]

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 LuckyPatcher anbietet[37], 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/modifcation 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

6.3 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 [37]

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 feasable, da einen mechanismus zu erstellen komplexer ist als die app itself

se/tee muss es eine lösung geben sonst braucht man für verschiedene apps verscheidene 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 Android's architecture [39] 9
2.2 apk [34] 10
2.3 dex [34] 12
2.4 java 12
2.5 install 13
2.6 oat 16
2.7 art 17
2.8 art2 18
2.9 oatdex 19
2.10 artarch 19
2.11 lvl [16] 21
2.12 amazon 24
2.13 amazonFolder 24
2.14 re1 27
2.15 re2 28

3.1 Left: Features offered LuckyPatcher
Middle: Variants to crack license verification
Right: Result after patching 36

4.1 tee [10] 64

5.1 tee [10] 75

List of Tables

3.1 Functionality for the test apps before and after patching 39
3.2 Overview of License Verification Library patching patterns applied by each modus 46

List of Code Snippets

2.1 Calling the LVL . . . 22
2.2 Calling the LVL . . . 22
2.3 Callback . . . 23
2.4 Callback . . . 25
2.5 Creating Zirconia . . . 26
2.6 Callback . . . 26
2.7 Name, Quelle . . . 29

3.1 Diff on Dex level for N1 pattern . . . 39
3.2 Diff on Smali level for N1 pattern . . . 39
3.3 Diff on Java level for N1 pattern (abstracted) . . . 40
3.4 Diff on Dex level for N2 pattern . . . 40
3.5 Diff on Smali level for n2 pattern . . . 40
3.6 Diff on Java level for N2 pattern (abstracted) . . . 40
3.7 Diff on Dex level for N3 pattern . . . 41
3.8 Diff on Smali level for N3 pattern . . . 41
3.9 Diff on Java level for N3 pattern (abstracted) . . . 41
3.10 Diff on Dex level for N4 patch . . . 42
3.11 Diff on Smali level for N4 patch . . . 42
3.12 Diff on Java level for N4 patch (abstracted) . . . 42
3.13 Diff on Dex level for N6 patch . . . 43
3.14 Diff on Smali level for N6 patch . . . 43
3.15 Diff on Java level for N6 patch (abstracted) . . . 43
3.16 Diff on Java level for N7 patch (abstracted) . . . 43
3.17 Diff on Dex level for Amazon patch . . . 44
3.18 Diff on Smali level for Amazon patch . . . 44
3.19 Diff on Java level for Amazon patch (abstracted) . . . 44
3.20 Diff on Dex level for Samsung patch . . . 45
3.21 Diff on Smali level for Samsung patch . . . 45
3.22 Diff on Java level for Samsung patch (abstracted) . . . 45

4.1 asd[15] . . . 49
4.2 Partial Listing . . . 50

4.3 Partial Listing 51
4.4 Partial Listing 53
4.5 Partial Listing 54

Bibliography

[1] allatori. *Allatori Java Obfuscator*. URL: http://www.allatori.com/ (visited on 01/22/2016).

[2] allatori. *Documentation*. URL: http://www.allatori.com/doc.html (visited on 01/22/2016).

[3] allatori. *ProGuard*. URL: http://developer.android.com/tools/help/proguard.html (visited on 01/22/2016).

[4] 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).

[5] 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).

[6] Android Developers. *Adding Licensing to Your App*. URL: https://developer.android.com/google/play/licensing/adding-licensing.html (visited on 01/18/2016).

[7] Android Developers. *Application Fundamentals*. URL: http://developer.android.com/guide/components/fundamentals.html (visited on 01/18/2016).

[8] Android Developers. *Licensing Overview*. URL: https://developer.android.com/google/play/licensing/overview.html (visited on 01/18/2016).

[9] Apple. *Piracy Prevention*. URL: http://www.apple.com/legal/intellectual-property/piracy.html (visited on 01/18/2016).

[10] ARM. *TrustZone*. URL: http://www.arm.com/products/processors/technologies/trustzone/index.php (visited on 01/24/2016).

[11] Blackmart. *Blackmart Alpha*. URL: http://www.blackmart.us/ (visited on 01/20/2016).

[12] 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).

[13] 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).

[14] 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 (visited on 01/24/2016).

[15] ChelpuS. *Lucky Patcher*. URL: http://lucky-patcher.netbew.com/ (visited on 01/09/2016).

[16] 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).

[17] 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).

[18] CrackAPK. *Android APK Cracked*. URL: http://www.crackapk.com/ (visited on 01/20/2016).

[19] A. Developers. *Licensing Reference*. URL: https://developer.android.com/google/play/licensing/licensing-reference.html (visited on 01/21/2016).

[20] 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).

[21] D. Ehringer. *The Dalvik Virtual Machine Architecture*. Mar. 2010.

[22] 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).

[23] Google Developers. *AdMob for Android*. URL: https://developers.google.com/admob/android/quick-start (visited on 01/26/2016).

[24] Google Play. *Google Play*. URL: https://play.google.com/store?hl=de (visited on 01/26/2016).

[25] GuardSquare. *DexGuard - The strongest Android obfuscator, protector, and optimizer*. URL: https://www.guardsquare.com/dexguard (visited on 01/22/2016).

[26] F. Guo, P. Ferrie, and T.-c. Chiueh. "A Study of the Packer Problem and Its Solutions." English. In: *Recent Advances in Intrusion Detection*. Ed. by R. Lippmann, E. Kirda, and A. Trachtenberg. Vol. 5230. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 98–115. ISBN: 978-3-540-87402-7. DOI: 10.1007/978-3-540-87403-4_6.

[27] IDC Research, Inc. *Smartphone OS Market Share*. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 01/16/2016).

[28] T. Johns. *Securing Android LVL Applications*. URL: http://android-developers.blogspot.de/2010/09/securing-android-lvl-applications.html (visited on 01/18/2016).

[29] 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).

[30] 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).

[31] A. Kovacheva. "Efficient Code Obfuscation for Android." Master's Thesis. Université de Luxembourg, Faculty of Science, Technology and Communication, Aug. 2013.

[32] 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-erstmals-groser-als-bei-apple/ (visited on 01/16/2016).

[33] J. Levin. *Android Security - New threats, New Capabilities*. URL: http://newandroidbook.com/files/Andevcon-Sec.pdfand (visited on 01/18/2016).

[34] J. Levin. *Dalvik and ART*. Dec. 2015.

[35] M. Liersch. *Android Piracy*. URL: https://www.youtube.com/watch?v=TNnccRimhsI (visited on 01/22/2016).

[36] J. F. R. Markus F.X.J. Oberhumer László Molnár. *Ultimate packer for eXecutables*. URL: http://upx.sourceforge.net/ (visited on 01/22/2016).

[37] M.-N. Muntean. "Improving License Verification in Android." Master's Thesis. Technische Universität München, Fakultät für Informatik, May 2014.

[38] R. Nigam. *Android Packers: Separating from the pack*. URL: http://www.fortiguard.com/uploads/general/Area41Public.pdf (visited on 01/22/2016).

[39] Obscure - community site theme. *Understanding the Android software stack*. URL: http://maat-portfolio.mut.ac.th/~r4140027/?p=116 (visited on 01/27/2016).

[40] Y. Park. *We can still crack you! - General unpacking method for Android packer(NO ROOT)*. URL: https://www.blackhat.com/docs/asia-15/materials/asia-15-Park-We-Can-Still-Crack-You-General-Unpacking-Method-For-Android-Packer-No-Root.pdf (visited on 01/22/2016).

[41] R. Price. *DexFile*. URL: http://www.businessinsider.com/android-app-profitability-v-ios-2015-1?IR=T (visited on 01/16/2016).

[42] 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).

[43] 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=T000000062L (visited on 01/19/2016).

[44] P. Schulz. "Code Protection in Android." Lab Course. Friedrich-Wilhelms-Universität Bonn, Institute of Computer Science, July 2012.

[45] M. T. Serrafero. *Piracy Testimonies, Causes and Prevention*. URL: http://www.xda-developers.com/piracy-testimonies-causes-and-prevention/ (visited on 01/16/2016).

[46] ST life.augmented. *Allatori Java Obfuscator*. URL: http://www.st.com/web/catalog/mmc/FM143/SC1282/PF259413 (visited on 01/24/2016).

[47] stack overflow. *Posts containing 'Android, Piracy'*. URL: http://stackoverflow.com/search?q=android+piracy (visited on 01/26/2016).

[48] stack overflow. *Posts containing 'Lucky Patcher'*. URL: http://stackoverflow.com/search?q=lucky+patcher (visited on 01/26/2016).

[49] 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).

[50] 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).

[51] 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).

[52] T. Strazzere. *APKfuscator*. URL: `https://github.com/strazzere/APKfuscator` (visited on 01/21/2016).

[53] T. Strazzere. *dehoser*. URL: `https://github.com/strazzere/dehoser` (visited on 01/21/2016).

[54] T. Strazzere. *Practicing Safe Dex*. URL: `https://www.youtube.com/watch?v=Rv8DfXNYnOI` (visited on 01/21/2016).

[55] T. Strazzere. *Practicing Safe Dex*. URL: `http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf` (visited on 01/21/2016).

[56] TeamSpeak Systems GmbH. *TeamSpeak 3*. URL: `https://play.google.com/store/apps/details?id=com.teamspeak.ts3client&hl=de` (visited on 01/20/2016).

[57] The University of North Caroline Greensboro. *Software Licensing*. URL: `http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/` (visited on 01/16/2016).

[58] J. S. Tim Strazzare. *Android Hacker Protection Level 0*. URL: `https://www.youtube.com/watch?v=6vFcEJ2jgOw` (visited on 01/22/2016).

[59] J. Underwood. *Today Calendar's Piracy Rate*. URL: `https://plus.google.com/+JackUnderwood/posts/jWs84EPNyNS` (visited on 01/16/2016).

[60] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. "AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware." English. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by H. Bos, F. Monrose, and G. Blanc. Vol. 9404. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 359–381. ISBN: 978-3-319-26361-8. DOI: `10.1007/978-3-319-26362-5_17`.

[61] R. Yu. *Android packer - facing the challenges, building solutions*. URL: `http://www.allatori.com/` (visited on 01/22/2016).