



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



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

Munich, March 15, 2015

Johannes Neutze, B. Sc.

Acknowledgments

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

Assumptions

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

Abstract

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

Contents

Acknowledgments	iii
Assumptions	iv
Abstract	v
Glossary	1
Acronyms	2
1 Introduction	3
1.1 Motivation	3
1.2 Related Work	4
2 Foundation	5
2.1 Software Piracy	5
2.1.1 Threat to Developers	5
2.1.2 Risks to Users	6
2.1.3 Piracy on Android	6
2.2 Android	7
2.2.1 Introduction	7
2.2.2 Dalvik Virtual Machine	8
2.2.3 Dalvik EXecutionable File	9
2.2.4 Build Process of Android Applications	13
2.2.5 ART	15
2.2.6 Installation, Running and Original Copy Protection	18
2.2.7 Root on Android	21
2.3 License Verification Libraries	21
2.3.1 Google	22
2.3.2 Amazon	25
2.3.3 Samsung	27
2.4 Code Analysis	28

3	Cracking Android Applications with LuckyPatcher	37
3.1	What is LuckyPatcher and what is it used for?	37
3.2	Modus Operandi	38
3.3	Variants for Cracking License Verification	40
3.4	Patching Patterns	40
3.5	Learnings from LuckyPatcher	48
4	Counter Measurements for Developers	50
4.1	Tampering Protection	50
4.1.1	Prevent Debuggability	51
4.1.2	Root Detection	52
4.1.3	LuckyPatcher Detection	53
4.1.4	Sideload Detection	54
4.1.5	Signature	55
4.1.6	Flow Control	56
4.2	Library Modifications	57
4.2.1	Modify the Library	57
4.2.2	Native Implementation	58
4.3	Reverse Engineering Prevention	58
4.3.1	Break Common Reengineering Tools	59
4.3.2	Obfuscation	60
4.3.3	Packers	63
4.4	External Support	64
4.4.1	Service-managed Accounts	65
4.4.2	ART endlich durchsetzen	65
4.4.3	Secure Elements	65
4.4.4	Trusted Execution Environment	66
5	Evaluation of Counter Measurements	67
5.1	Tampering Protection	67
5.2	Library Modifications	67
5.2.1	Modify the Library	67
5.2.2	Native Implementation	68
5.3	Reverse Engineering Prevention	69
5.3.1	Break Common Reengineering Tools	69
5.3.2	Obfuscation	70
5.3.3	Packers	73
5.3.4	Packers	75

Contents

5.4	External Support	75
5.4.1	Service-managed Accounts	75
5.4.2	ART endlich durchsetzen	76
5.4.3	Secure Elements	76
5.4.4	Trusted Execution Environment	76
6	Conclusion	78
6.1	Summary	78
6.2	Discussion	78
6.3	Future Work	79
	List of Figures	81
	List of Tables	82
	List of Code Snippets	83
	Bibliography	85

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.

.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.

.odex Optimized Dalvik EXecutable file.

ADB Android Debug Bridge.

ADT Android Developer Tools.

AOT Ahead-Of-Time.

API Application Programming Interface.

APK Android Application Package.

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.

1 Introduction

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 [50].

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[51].

1.1 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 [14][28]. This giant number of Android devices is powered by Google Play [25], 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 [58]. 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 [33].

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[42]. 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 [60] and express their need to protect their IP on platforms like xda-developers [46] or stackoverflow [48]. Many of the developers having problems with the license verification mechanism name Lucky Patcher as one of their biggest problems [49].

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

1.2 Related Work

Since piracy on Android is a serious topic, there has been research in this field already.

The Master's Theses of Patrick Bernhard [11] and Marius-Nicolae Muntean [32] both analysed attacks on different license verification libraries. This included the solutions of Amazon, Google and Xamarin. The authors give an overview of possible solutions to protect the application by using techniques like obfuscation.

A step further is the Master's Thesis of Alexandrina Kovacheva. She evaluates applications from the store for their protection mechanisms and then suggests an implementation of an bytecode obfuscator.

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 [24]. The Ad Unit ID is responsible to assign earnings generated by an mobile advertisement to the developer. When an application is pirated, this code can be replaced

by the pirate's one. Future revenues generated by advertisements 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 malicious 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 application. 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 [36].

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.[13][36]

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[21]. 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 [12], and websites for cracked Android

Application Package (APK)s, as crackApk[19]. 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 programm.

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. [60][46]

For this reason some developers do not implement any copy protection at all since it is cracked withing days [30]. Especially Android applications are at high risk for piracy due bytecode in general is an easy target to reverse engineer.

2.2 Android

2.2.1 Introduction

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

Android is an open source Linux-based operating system running on a large set of touchscreen devices, wearables and many more

Launched in 2007 by Google, it is designed to meet the limited computational capacity of a mobile device's hardware. The principal processor of Android devices is the ARM platform for which the operating system is optimized

AUFBAU ANDROID The underlying entity of the system is its kernel which bridges the hardware of the device and the remaining software components. Being a Linux-based kernel, it allows remote access to the device via a Linux shell as well as the execution of standard Unix commands.

A level above is the Android Runtime, which will be explained closer

At the same abstraction level as the virtual machine are the native libraries of the system. Written in C/C++, they permit low level interaction between the applications and the kernel through Java Native Interface (JNI)

The next layer is the application framework which provides generic functionality to mobile software through Android's API. The top layer of the Android OS stack is where custom applications are compiled, installed and executed.

[32]

What is Android? Where is it used? When was it founded? Who does it belong to?

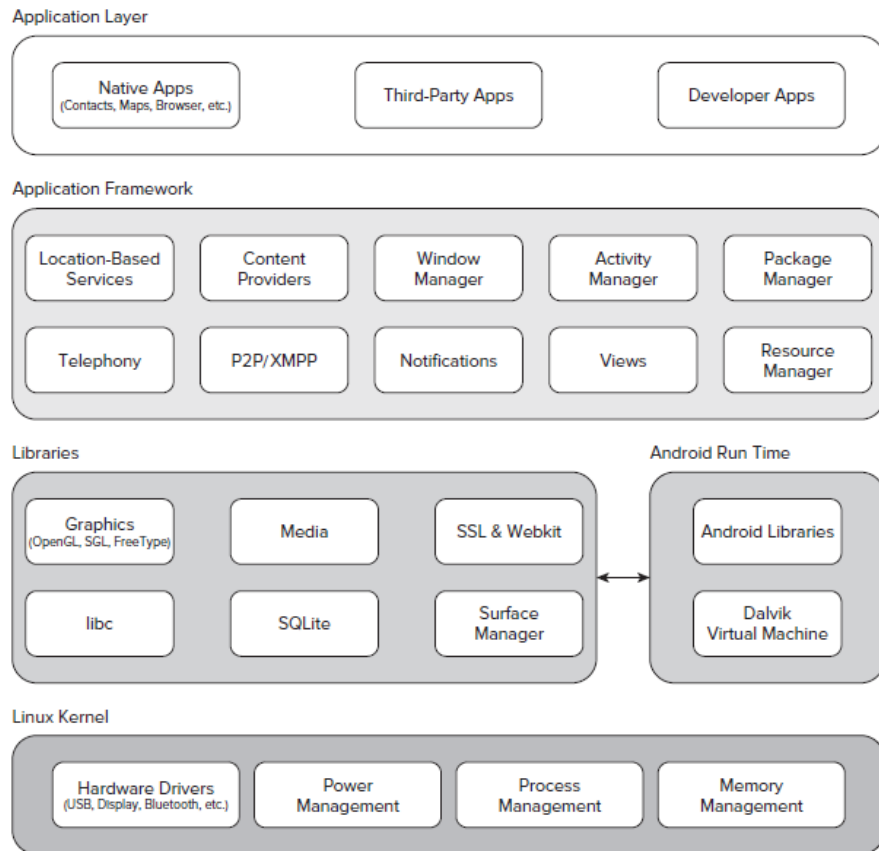


FIGURE 1-1

Figure 2.1: stack

platform developed by the "Android Open Source Project" -see- official website?
 currently one of the main operating systems for mobile devices -see- quelle
 focussed on simplicity for users, install of apps etc -see- quelle
 riesiger markt

2.2.2 Dalvik Virtual Machine

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

Flow what is dalvik, what is different to java, what is dex (build process), VGL
<http://newandroidbook.com/files/ArtOfDalvik.pdf>

The applications for Android are written using the Java programming language.

stack abstraction is the Dalvik Virtual Machine (DVM)

DVM is highly tailored to work according to the specifications of the Android platform optimized for a slower CPU in comparison with a stationary machine and works with relatively little RAM memory (• limited processor speed • limited RAM • no swap space • battery powered • diverse set of devices • sandboxed application runtime)[22] DVM is register-based, differing from the standard Java Virtual Machine (JVM) which is stack-based, register-based architectures require fewer executed instructions than stack-based architectures, register-based code is approximately 25 percent larger than the stack-based, the increase in the instructions fetching time is negligible: 1.07 percent extra real machine loads[22]

the Android OS has no swap space imposing that the virtual machine works without swap. Finally, mobile devices are powered by a battery thus the DVM is optimized to be as energy preserving as possible, Except being highly efficient, the DVM is also designed to be replicated quickly because each application runs within a “sandbox”: a context containing its own instance of the virtual machine assigned a unique Unix user ID

wie der build process funktioniert wird später gesondert beschrieben, hier sagen wir einfach das ergebnis ist die dex datei

[32] [22]

DVM is - Customized optimized JVM based on Apache Harmony, - Not fully J2SE or J2ME compatible -see- Java compiles into DEX code -see- 16-bit opcodes and Register, rather than stack-based

History, Dalvik was introduced along with Android, created by Created by Dan Bornstein, Named after an Icelandic town, 2.2 brought current just in time compilation (ERKLÄREN)

Dalvik vs Java - Dalvik is a virtual machine implementation, Based on Apache Harmony, Borrows heavily from Java - Brings significant improvements over Java, in particular J2ME, Virtual Machine architecture is optimized for memory sharing, Reference counts/bitmaps stored separately from objects, Dalvik VM startup is optimized through Zygote - Java .class files are further compiled into DEX.

Overview creating APK unterschied zu java und dann auf dex?

[35]

2.2.3 Dalvik EXecutionable File

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

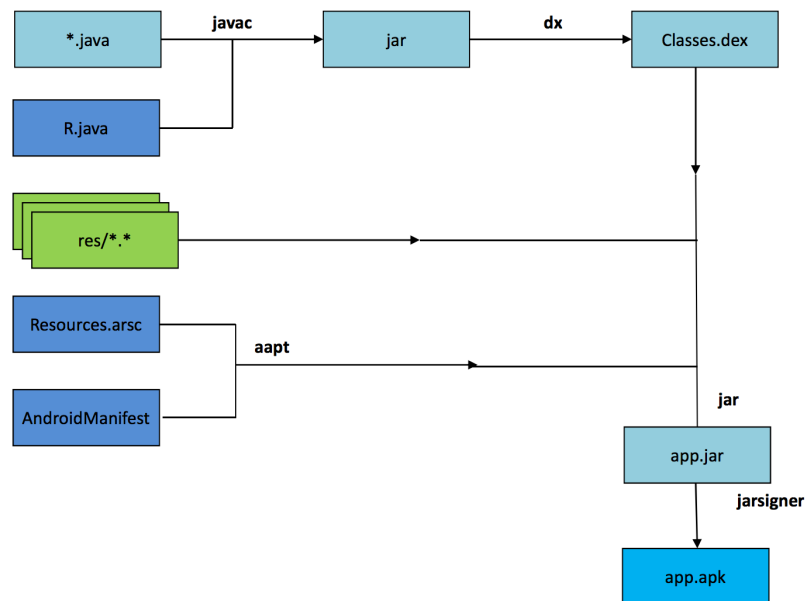


Figure 2.2: apk

AUFBAU DEX DVM is register based. Registers are considered 32 bits wide to store values such as integers or floating point numbers. Adjacent register pairs are used to store 64-bit values

dest-then-source ordering for its arguments

there are 218 used valid opcodes in Dalvik bytecode -see- QUELLE

Due to its simplicity over bytecode for other architectures as well as the little protection applied in practice, Dalvik bytecode is currently an easy target for the reverse engineer.

Each .class file has its own heterogeneous constant pool which may contain duplicating data, BEISPIEL, memory efficiency of a .dex file comes primarily from the type-specific constant pools used to store the data, BEISPIEL, significantly more references within a .dex file compared to a .class file[22] compression as efficiently as up to 44 percent of the size of an equivalent .jar archive[22] hier noch einfach dex, später erst opcode nennen, the exact dex format will be explained in 2.4.2

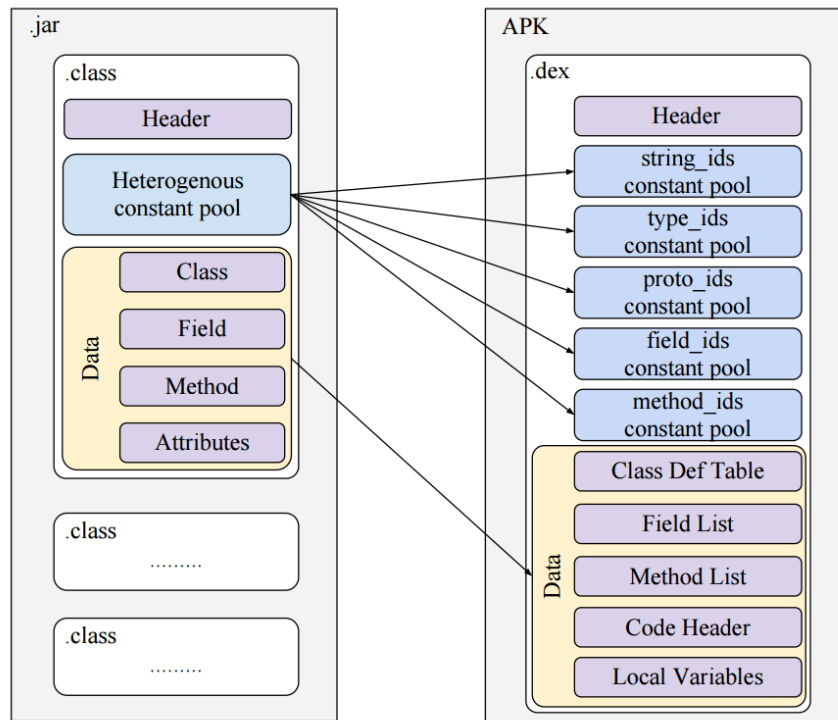


Figure 2.3: java

[32] [22]

dx utility converts multiple class files to classes.dex, java bytecode is converted to dex bytecode, dex instructions 16bit multiples, java 8bit constant, string, type and method pools are merged, significant savings for strings, types and methods in multiple classes overall memory footprint diminished by about 50% dex file format specified in android documentation -see- SOURCE

dex instructions refer to indexes (in pools)

dex bytecode is strictly similar to java bytecode, allows for ease of/re compilation back and forth to/from java

dex vs java

java vm is stack based, dex is register based java bytecode is actually more compact than dex dex bytecode is more suited to arm architectures, mapping from dex registers to arm registers dex supports bytecode optimizations, java no, apk's classes.dex are optimized before install, on device

The DEX file format

	Magic	DEX Magic header ("dex\n" and version ("035 "))
Adler32 of header (from offset +12)	checksum	SHA-1 hash of file (20 bytes)
	signature	
Total file size	File size	Header size (0x70)
0x12345678, in little or big endian form	Endian tag	Unused (0x0)
Unused (0x0)	Link offset	Map offset
Number of String entries	String IDs Size	String IDs offset
Number of Type definition entries	Type IDs Size	Type IDs offset
Number of prototype (signature) entries	Proto IDs Size	Proto IDs offset
Number of field ID entries	Field IDs Size	Field IDs offset
Number of method ID entries	Method IDs Size	Method IDs offset
Number of Class Definition entries	Classdef IDs Size	Classdef IDs offset
Data (map + rest of file)	Data Size	Data offset

Figure 2.4: dex1

[35]

AUFBAU DALVIK BYTECODE

The program code of an Android application is delivered in form of Dalvik bytecode. It will be executed by the Dalvik Virtual Machine and is comparable to Java bytecode. So there is a separate optimizing step while installing an application, where the bytecode gets optimized for the underlying architecture. The optimized form is also called "odex". The optimization is done by a program called "dexopt" which is part of the Android platform. The DVM can execute optimized as well as not optimized Dalvik bytecode.

unterschied <http://newandroidbook.com/files/Andevcon-DEX.pdf>

.dex file The Dalvik bytecode consists of opcodes and is thus hard to read for humans. The Cracking Tool has to modify the opcodes in order to alter the behaviour of the application. Since it is directly read by the Dalvik virtual machine, it is the single point of truth.

.odex file Dalvik bytecode of an application is normally not optimized, because it is executed by a DVM which can run under different architectures.

Magic		Type	Implies	Size	Offset
checksum		0x0	DEX Header	1 (implies Header Size)	0x0
signature		0x1	String ID Pool	Same as String IDs size	Same as String IDs offset
File size	Header size	0x2	Type ID Pool	Same as Type IDs size	Same as String IDs offset
Endian tag	Link size	0x3	Prototype ID Pool	Same as Proto IDs size	Same as ProtoIDs offset
Link offset	Map offset	0x4	Field ID Pool	Same as Field IDs size	Same as Field IDs offset
String IDs Size	String IDs offset	0x5	Method ID Pool	Same as Method IDs size	Same as Method IDs offset
Type IDs Size	Type IDs offset	0x6	Class Defs	Same as ClassDef IDs size	Same as ClassDef IDs offset
Proto IDs Size	Proto IDs offset	0x1000	Map List	1	Same as Map offset
Field IDs Size	Field IDs offset	0x1001	Type List	List of type indexes (from Type ID Pool)	
Method IDs Size	MethodIDs offset	0x1002	Annotation set	Used by Class, method and field annotations	
Classdef IDs Size	Classdef IDs offset	0x1003	Annotation Ref		
Data Size	Data offset	0x2000	Class Data Item	For each class def, class/instance methods and fields	
		0x2001	Code	DexCodeItems – contains the actual byte code	
		0x2002	String Data	Pointers to actual string data	
		0x2003	Debug Information	Debug_info_items containing line no and variable data)	
		0x2004	Annotation	Field and Method annotations	
		0x2005	Encoded Array	Used by static values	
		0x2006	Annotations Directory	Annotations referenced from individual classdefs	

Figure 2.5: dex2

2.2.4 Build Process of Android Applications

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

APK The file format of the install ready application is called Android Package (APK) and all the mobile software is distributed over Google Play in this format. The APK format is a package management system based on the ZIP file archive format.

CONTENT: 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

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 Asset-Manager.

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

resources.arsc: a file containing precompiled resources, such as binary XML for example.

BUILD PROCESS written using the Java programming language.

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.

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

[32] [22]

für 4.1 erklären, bild <http://developer.android.com/tools/building/index.html>
<http://newandroidbook.com/files/ArtOfDalvik.pdf> seite 10

Aufbau APK erklären

many steps and tools until the APK is build and ready to be deployed
applications are written in the Java programming language by the developer, code is available in .class files

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

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)

step 3: packing and signing the APK, ApkBuilder constructs an apk file out of the "classes.dex" file and adds further resources like images and ".so" files, ".so" files are shared objects which contains native functions that can be called from within the DVM, jarsigner adds developers signature to APK(ERKLÄREN WARUM SIGNATURE NÖTIG UND WO GEPRÜFT), now the app can be installed

Android applications are written in the Java [11] programming language and deployed as files with an ".apk" suffix, later called APK. It is basically a ZIP-compressed file and contains resources of the application like pictures and layouts as well as a dex file

This dex file, saved as "classes.dex", contains the program code in form of Dalvik bytecode. Further explanations on this bytecode format are given in section 3.2. The content of the APK is also cryptographically signed, which yields no security improvement but helps to distinguish and confirm authenticity of different developers of Android applications.

Die apk kann dann per adb, market oder direkt installiert werden

Within the installation process, every installed application gets its own unique user ID (UID) by default. This means that every application will be executed as a separate system user. -see- QUELLE/SINN?

so why is dalvik deprecated? JIT is slow, consuming both cycles and battery power garbage collection causes hangs/jitters dalvik is 32bit, cannot benefit from 64bit architecture kitkat first introduce art, lollipop adopts it

2.2.5 ART

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 minor versions, android optimizes apps everytime you update

art was designed to address shortcomings of dalvik: virtual machine maintenance is expensive, interpreter/jit simply aren't efficient as native code, doing jit all over again on every execution is wasteful, maintenance threads require significantly more 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 is following ios into 64bit space

advantages of art art moves compilation from Just-In-Time (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 parallelizable (foreground/background), non blocking -see- QUELLE 64bit but some issues still exist -see- quelle

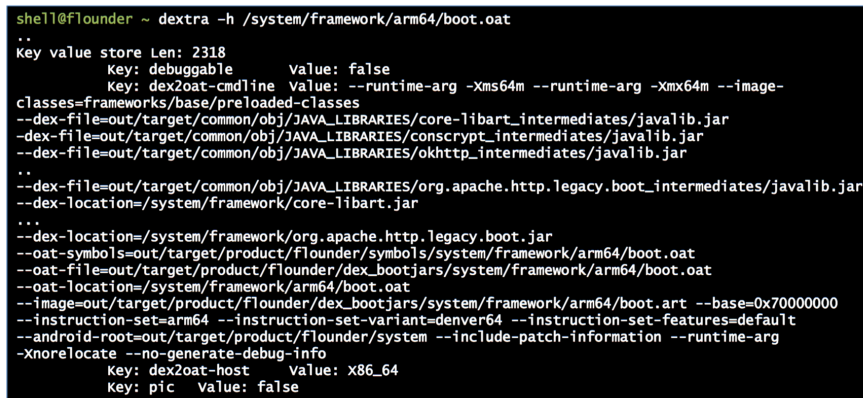
main idea of art/aot: actually compilation can be to one of two types, quick (native code), portable(llvm code) in practice preference is to compile to native code, portable implies another layer of IR (LLVM's bit code)

Art itself: art uses not one but two file formats art - only one file, boot.art, in /system/framework/architecture (arm,...) oat - master file, boot.oat, in /system/framework/architecture (arm,...) - odex files no longer optimized dex but oat, alongside apk for

system apps/frameworks, /data/dalvik-cache for 3rd party apps, still uses odex extension, now file format is elf/oat

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



```
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/javali.b.jar
--dex-file=out/target/common/obj/JAVA_LIBRARIES/conscrypt_intermediates/javali.b.jar
--dex-file=out/target/common/obj/JAVA_LIBRARIES/okhttp_intermediates/javali.b.jar
..
--dex-file=out/target/common/obj/JAVA_LIBRARIES/org.apache.http.legacy.boot_intermediates/javali.b.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

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

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

2 Foundation

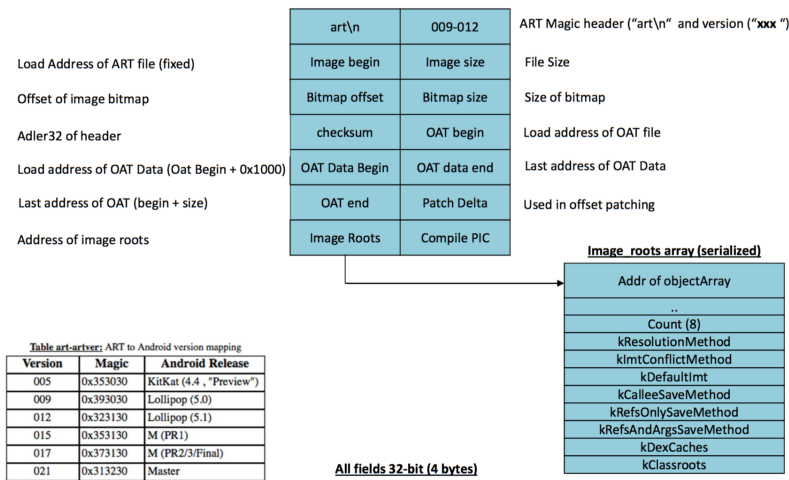


Figure 2.7: art

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, likewise 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

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

[35]

Lollipop (5.x)		Marshmallow (PR1)		Marshmallow (PR2-Release)	
art\n	009-012	art\n	015	art\n	017-???
Image begin	Image size	Image begin	Image size	Image begin	Image size
Bitmap offset	Bitmap size	ART Fields Offset	ART Fields Size	OAT checksum	OAT begin
checksum	OAT begin	Bitmap offset	Bitmap size	OAT Data begin	OAT Data end
OAT Data begin	OAT data end	checksum	OAT begin	OAT end	Patch Delta
OAT end	Patch Delta	OAT Data begin	OAT data end	Image Roots	Size of Pointer
Image Roots	Compile PIC	OAT end	Patch Delta	Compile_pic	Objects Offset
		Image Roots	Compile PIC	Objects Size	Fields Offset
				Fields Size	Methods offset
				Methods size	Strings Offset
				Strings size	Bitmap offset
				Bitmap size	

... Followed by Image Roots

All fields 32-bit (4 bytes)

Figure 2.8: art2

2.2.6 Installation, Running and Original Copy Protection

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

Installation two steps: primary is the APK verification and secondary is the bytecode optimization

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

ablauf starten von app

When an Android application is executed, the process consists of the following four parts: • Dalvik bytecode, which is located in the dex file • Dalvik Virtual Machine

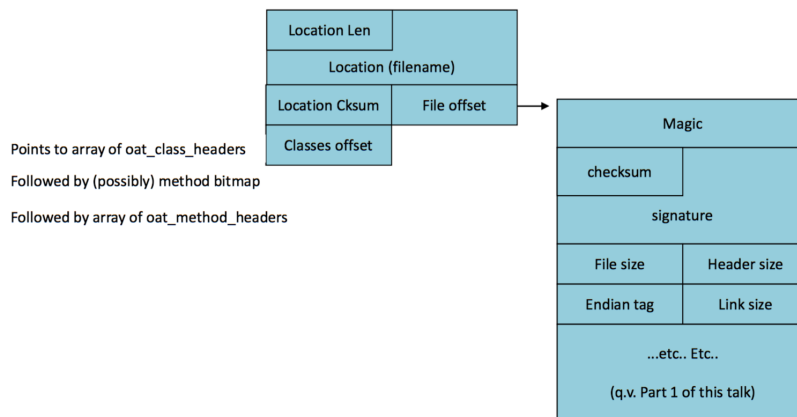


Figure 2.9: oatdex

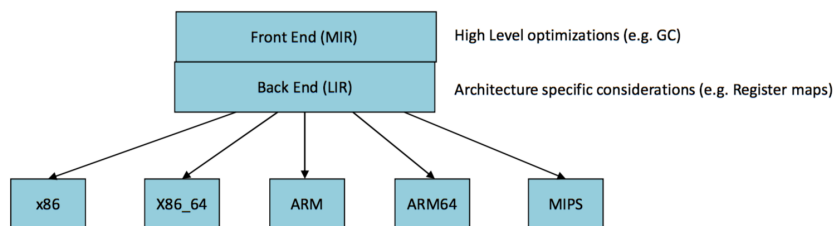


Figure 2.10: artarch

[13], which executes the Dalvik bytecode • Native Code, like shared objects, which is executed by the processor • Android Application Framework, which provides services for the application [32]

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 copied 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

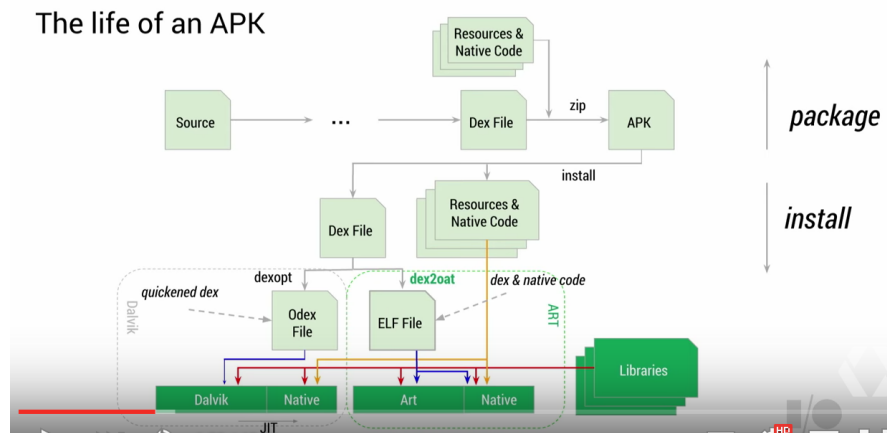


Figure 2.11: install

started optimized code from dalvik cache will be executed instead of apk [39]

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]

old system was to put APK in folder user cannot access, unless you root weak, almost non-existent, 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 was almost non-existent.

kann mit root umgangen werden

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

2.2.7 Root on Android

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

what is rooting? 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, 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

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

examples modify system variables, e.g. to utilize the notification led on motorola devices which is usually disabled install custom roms [38]

android's content protection are invalid when rooted any application's data directory (or code) can be read DRm can be bypassed coupled with dex decompilkation big problem, app can be decompiled, modded and repackaged [34]

a list of root vulnerabilities can be found on <http://androidvulnerabilities.org/all>

2.3 License Verification Libraries

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

Since the original approach of subsection 2.2.6 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 [39]

put copy protection methods into app itself, kind of DRM

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

2.3.1 Google

gradThis 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
[39]

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) [31]

introduced to fight piracy in Google Play, introduced 07/27/2010 simple and free
service [17]

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 information: - package
name of app, a nonce that has to be present in every server response to ensure attacker
security, callback for async handling of server response, implemented in initial license
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 signs response with app private key public key used to
check signature of response, effective mechanism is established to ensure origin and
detect tampering [39]

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

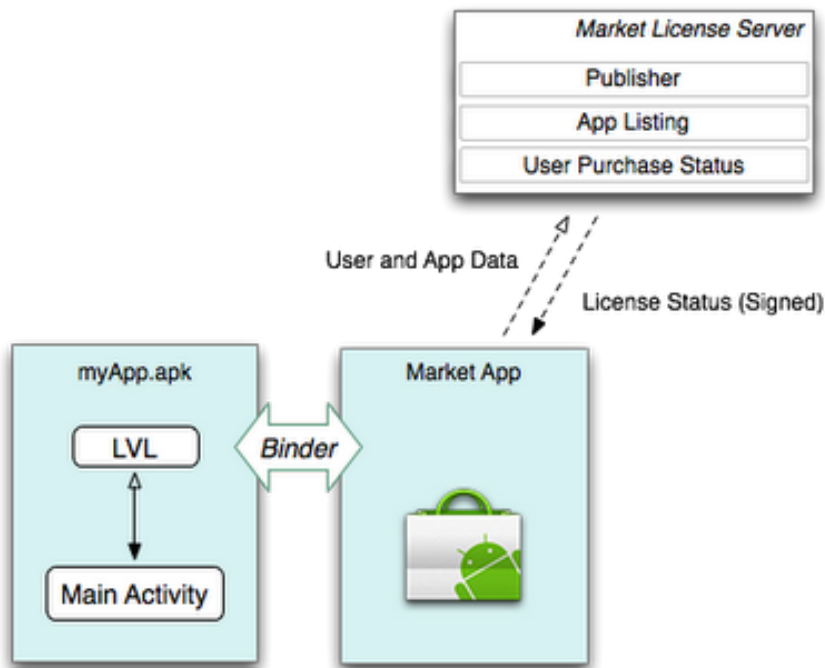


Figure 2.12: lvl [17]

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 licensign 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 ap-
plications 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 access to Google Play. Access is not bound to characteristics of host device, but Google Play and licensing policy defined can be installed, managed 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 records real time over network [17]

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

Code Snippet 2.2: Calling the LVL

```
57  @Override
58  public void allow(final int reason) {
```

```

59     ...
60 }
61
62 @Override
63 public void dontAllow(final int reason) {
64     ...
65 }
66
67 @Override
68 public void applicationError(final int errorCode) {
69     ...
70 }

```

Code Snippet 2.3: Callback

2.3.2 Amazon

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, alternative 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

Apply Amazon DRM? * ☒ Yes (Recommended) ☐ No
Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.

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 2.13: amazon

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

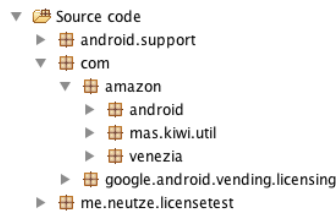


Figure 2.14: amazonFolder

on their own, done by amazon packaging tool when submitting can check amazon DRM (see picture), apply 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 developer before submitting is not necessary, amazon decompiles apk, injects drm code, compiles it and signs it with the "amazon developer" certificate [39]

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 classes, 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

[39]

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

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

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 [18] Galaxy Apps by Samsung, formerly known as Samsung Apps, for devices by Samsung renamed in July 2015 called zirconia for android [44]

Functional Principle

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

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

interior process: makes query for stored licensetest if found, app can execute if not exist or invalid, information of device and application will be 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 [44]

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

[44]

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

Code Snippet 2.5: Creating Zirconia

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

Code Snippet 2.6: 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. [32]

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

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

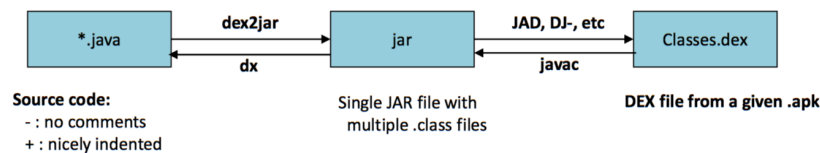


Figure 2.15: re1

like ads, registration, uncover sensitive data (app logic, secrets), replace certain classes with others (malicious ones), recompile back to jar, then dex, put cloned/trojaned version of your app on play or other market [35]

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

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

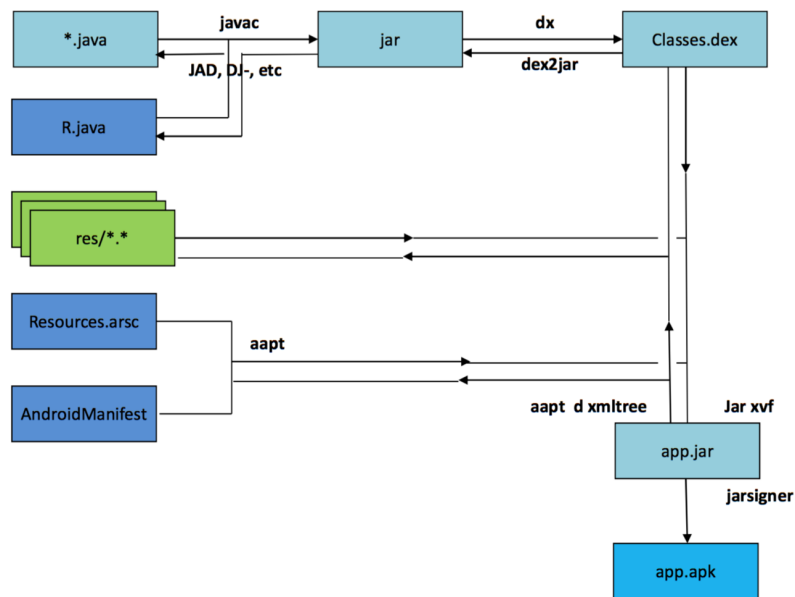


Figure 2.16: re2

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 Android Debug Bridge (ADB) tools installed
adb shell pm list packages -f(1) outputs a list of installed apps in formatf <name-space>.<appName> and an appended number "-1", each is a folder of installed app containing a base.apk (the application apk) example 2 enthält return von (1)

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

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

[39]

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

In order to analyse an APK, it has to be pulled from the Android device onto the computer. First the package name of the app has to be found out. This can be done by using the ADB. Entering example 1 returned example 2

dann auf verschiedenen leveln anschauen mit den folgenden tools

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

[35] 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) [32]

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 mini-

mizes 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 disassemblers 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

[32]

<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 funktioniert

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 device cracking application LuckyPatcher, especially on its license verification bypassing mechanism. reason lucky ptcher is taken extremely popular and easy(no technical knowdlege except root but apps can be traded) to use and discusses a lot in android community developeprs/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

[39]

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[52] 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 getestet 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,

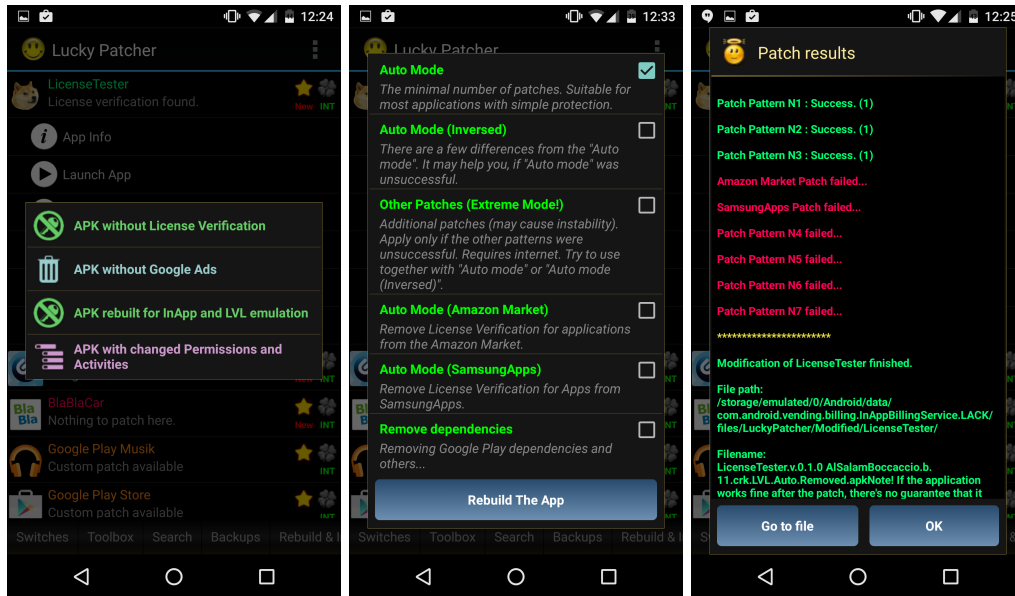


Figure 3.1: Left: Features offered LuckyPatcher
Middle: Variants to crack license verification
Right: Result after patching

bypass premium applications license verification, and more. To use all features, you need a rooted device." [16]

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 understand (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 obfuscation and anti-decompilation measure

so look at cracked application in order to understand 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_LICENSE 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 android

luckypatcher extracts original classes.dex from apk and patches it patching is done on binary level and done by using different patterns which are dependent on the modus (see patterns/modi) the result is either used to substitute 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

[39]

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 dateien 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 luckypatcher 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[43], Version 6.3, the created LicenseTest and Teamspeak 3[57], 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[20]

```
@@ 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[20] 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[20]

```
@@ 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[20]

```
@@ 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 aufgerufen 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 condition 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[20] 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 two 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

Modus	Patterns							
	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 occurred 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 dateien 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 [39]

4 Counter Measurements for Developers

Now 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 acquired 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 developer's 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 interfere 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 harder for attackers

CHARACTERIZED 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 [39]

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. [39]

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 [29]

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 funktionssweise 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

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

Code Snippet 4.1: asd[16]

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-licen>

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

Code Snippet 4.2: 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>

```
9  public static boolean checkInstall(final Context context) {
10      boolean result = false;
11      String[] luckypatcher = new String[]{
12          // Lucky patcher
13          "com.dimonvideo.luckypatcher",
14          // Another lucky patcher
15          "com.chelpus.lackypatch",
16          // Black Mart alpha
17          "com.blackmartalpha",
18          // Black Mart
19          "org.blackmart.market",
20          // Lucky patcher 5.6.8
21          "com.android.vending.billing.InAppBillingService.LUCK",
22          // Freedom
23          "cc.madkite.freedom",
24          // All-in-one Downloader
25          "com.allinone.free",
26          // Get Apk Market
27          "com.repodroid.app",
28          // CreeHack
29          "org.creepays.hack",
30          // Game Hacker
31          "com.baseappfull.fwd"
32      };
33
34      for (String string : luckypatcher) {
35          if(checkInstallerName(context, string)){
```

```
36         result = true;
37     }
38
39     if (result) {
40         android.os.Process.killProcess(android.os.Process.myPid());
41     }
42 }
43
44 return result;
45 }
46
47 private static boolean checkInstallerName(Context context, String string) {
48     PackageInfo info;
49     boolean result = false;
50
51     try {
52         info = context.getPackageManager().getPackageInfo(string, 0);
53
54         if (info != null) {
55             android.os.Process.killProcess(android.os.Process.myPid());
56             result = true;
57         }
58
59     } catch (final PackageManager.NameNotFoundException ignored) {
60     }
61
62     if (result) {
63         android.os.Process.killProcess(android.os.Process.myPid());
64     }
65     return result;
66 }
67 }
```

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-from-trusted-sources>
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

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

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. [39] !!!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?

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

Code Snippet 4.5: 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 [39]

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 modifications 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 [29] 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 way so 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 statement 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 function 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 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 Java Native Interface (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 application

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

[32] 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 techniqies to make it as hard a possible [39]

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[53] 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"[55] [56] it is a generic obfuscator and munger which works on dex files directly simple ideas the prototype 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 disasembly 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 [45]

use bad opcode in deadcode
e.g. reference not initied strings [56]

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 [56], 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 programmes 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

[32]

a few dex obfuscators exist, with different approaches proguard or sdex, rename methods, field and class 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 loops (minor impact of performance) can also use goto into other instructions

[35]

layout obfuscation most programmers name their variables, methods and classes in meaning full 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 identifiers that original meaning is lost while preserving correctness of syntax and semantics result is bytecode can be interpreted but dissable and decompile 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

[39]

will not protect against automated 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 [29]

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 [32]

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

[39]

identifier mangling, ProGuard uses a similar approach. It uses minimal lexical-sorted strings like a, b, c, ..., aa, ab, original identifiers give information about interesting parts of a program, Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed -see- neutralizing these information in order to prevent this reduction, remove any meta information about the behavior, meaningless string representation holdin respect to consistence means identifiers for the same object must be replaced by the same string, advantage of minimizing the memory usage, e development process in step "a" or step "b" string obfuscation, string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog, information is context, other is information itself, e.g. key, url, injective function and deobfuscation stub which constructs original at runtime so no behaviour is changed, does not make understanding harder since only stub is added but reduces usable meta information [45]

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 optimizations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [23] [59]

Dexguard

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

commercial Android obfuscator for bytecode and source code various techniques including strings encryption, encrypting app resources, tamper detection, removing logging code [32]

son of proguard, does everything that proguard does its a optimizer and shrinker, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), application tamper protection, removes debug information may increase dex size, memory size; decrease speed [59]

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 for android protects code: obfuscation, hides sensitive strings, keys and entire algorithms [26]

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 decrease in dex size, memory, increases speed, removed debug code like Proguard+string encryption [59]

commercial product from Smardec addition to what Proguard does it offers 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 obfuscation

[1] [2]

4.3.3 Packers

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

goal of packers is to protect the code from modifications or attacks, good or bad nature, often used by malware as well [61]

launcher 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 [37], 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 [27]

packer transforms code that a reverse engineer cannot directly extract information, e.g. by encrypting program data no information can be extracted without decrypting, would be bad for program as well that is why packer uses loader stub to decrypt in runtime, solution 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

BUILD 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 [45] step 4 execute

[41][62]

[45] 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
[39]

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[56], 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 dexClassLoader 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 bangle or IJIAMI, often for malware[61][62]

[40][54][59]

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 install 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 [59][41][62]

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 reengineering, 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!

best would be to have a security mechanism lucky patcher cannot access, since until now every android version could be rooted using an exploit thus very likely that it will be possible in the future as well all data thus can be accessed by the user in order to achieve secure storage introduce secure elements for critical code like license verification secure space separated from the device and its restrictions and vulnerabilities[kannengie\IeC {\ss }erProposal]

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 [47] 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

<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 privileged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitive 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

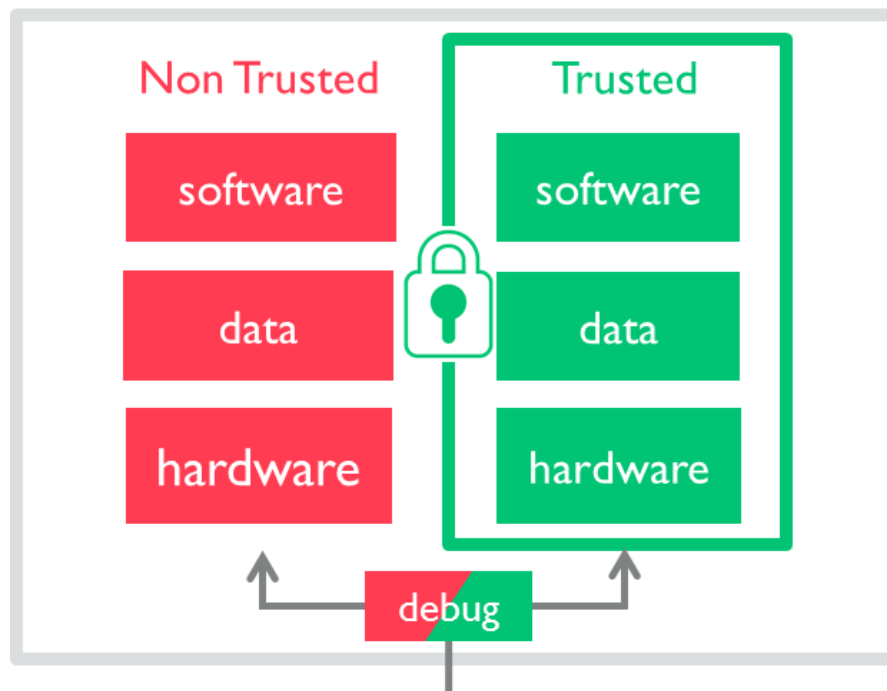


Figure 4.1: tee [10]

[15][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 attacker 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 combined 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 seems 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 functionality

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 application

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

[32] 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 techniques to make it as hard as possible [39]

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[53] 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"[55] [56] it is a generic obfuscator and munger which works on dex files directly simple ideas the prototype 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 [45]

use bad opcode in deadcode

e.g. reference not initied strings [56]

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 [56], 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 programmes 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

[32]

a few dex obfuscators exist, with different approaches proguard or sdex, rename

methods, field and class 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 loops (minor impact of performance) can also use goto into other instructions

[35]

layout obfuscation most programmers name their variables, methods and classes in meaningful way are preserved in generation of bytecode for JVM, hence still in dex, can be extracted by attacker, gain information and benefit when reengineering mangles names and identifiers that original meaning is lost while preserving correctness of syntax and semantics result is bytecode can be interpreted but disabled and decompiled 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

[39]

will not protect against automated 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 [29]

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 a 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 [32]

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

[39]

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

string obfuscation, a 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

[45]

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 optimizations removes debug information renames objects restructures code removes line numbers, stacktrace annoying [23] [59]

Dexguard

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

commercial Android obfuscator for bytecode and source code various techniques including strings encryption, encrypting app resources, tamper detection, removing logging code [32]

son of proguard, does everything that proguard does its a optimizer and shrinker, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), application tamper protection, removes debug information may increase dex size, memory size; decrease speed [59]

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 for android protects code: obfuscation, hides sensitive strings, keys and entire algorithms [26]

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 [59]

commercial product from Smardec addition to what Proguard does it offers 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 obfuscation

[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[61]

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 [37], 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 [27]

packer transforms code that a reverse engineer cannot directly extract information, e.g. by encrypting program data no information can be extracted without decrypting, would be bad for program as well that is why packer uses loader stub to decrypt in runtime, solution 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 [45] step 4 execute [41][62]

[45] 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 [39]

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[56], easy to spot since dex header is not normal static analysis will not work since encrypted file will be ignored, which contains 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 dexClassLoader 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 bangle or IJIAMI, often for malware[61][62]

[40][54][59]

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 [59][41][62]

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-malware.html>

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!

best would be to have a security mechanism lucky patcher cannot access, since until now every android version could be rooted using an exploit thus very likely that it will be possible in the future as well all data thus can be accessed by the user in order to achieve secure storage introduce secure elements for critical code like license verification secure space separated from the device and its restrictions and vulnerabilities[kannengie\IeC {\ss }erProposal]

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 [47] 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

<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 privileged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitive 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

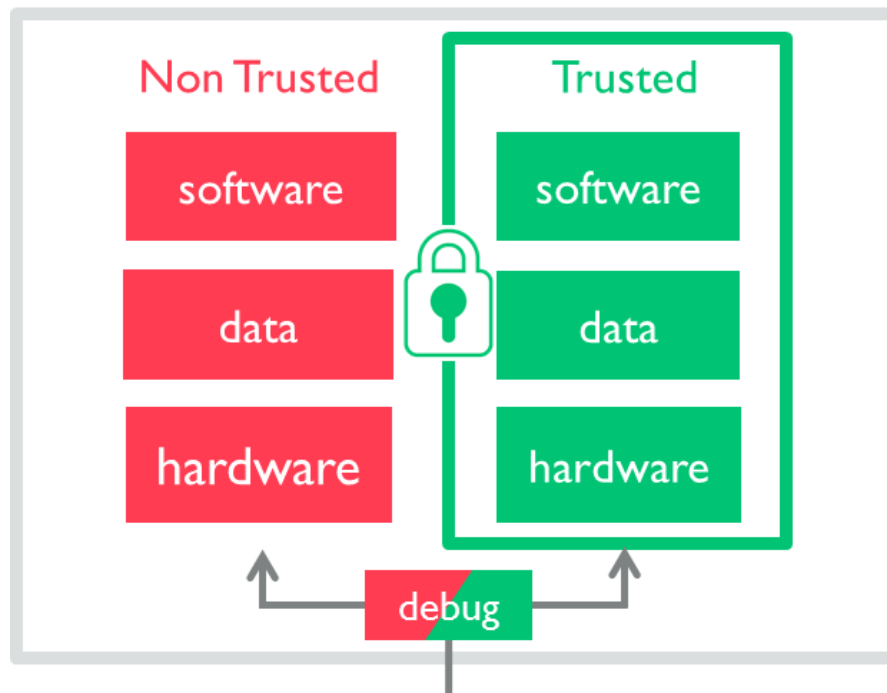


Figure 5.1: tee [10]

example Trustzone, Knox
[15][10]

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

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.

[45]

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

lvl popular but broken, has not done much since beginning of known issues [39]

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 resilience against reverse engineering, not exclusively to lvl thus shift from lvl protection to general protection against reverse engineering, decompilation and patching

eternal arms race no winning solution against all cases, just small pieces quantitative improvement no qualitatively improve resilience limited to quantitative resilience, matter of time until small steps generate more work for reengineering, ggf lower motivation for cracker only matter of time until patching tools catch up, completely new protection schemes need to be devised to counter those [39]

not a question of if but of when bytecode tool to generate the license library 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 [31]

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

alles hilft gegen lucky patcher auf den ersten blick, jedoch custom patches, welche LuckyPatcher anbietet[39], können es einfach umgehen, deswegen hilft nur reengineering schwerer zu machen viele piraten sind nicht mehr motiviert wenn es zu schwer ist every new layer of obfuscation/modification adds another level complexity

solange keine bessere lösung vorhanden unique machen um custom analysis und reengineering zu enforzen und dann viele kleine teile um die schwierigkeit des reengineerens 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 [39]

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

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

google hat schon sowas wie google vault

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

List of Figures

2.1	stack	8
2.2	apk	10
2.3	java	11
2.4	dex1	12
2.5	dex2	13
2.6	oat	16
2.7	art	17
2.8	art2	18
2.9	oatdex	19
2.10	artarch	19
2.11	install	20
2.12	lvl [17]	23
2.13	amazon	25
2.14	amazonFolder	26
2.15	re1	29
2.16	re2	30
3.1	Left: Features offered LuckyPatcher Middle: Variants to crack license verification Right: Result after patching	38
4.1	tee [10]	66
5.1	tee [10]	77

List of Tables

3.1	Functionality for the test apps before and after patching	41
3.2	Overview of License Verification Library patching patterns applied by each modus	48

List of Code Snippets

2.1	Calling the LVL	24
2.2	Calling the LVL	24
2.3	Callback	24
2.4	Callback	26
2.5	Creating Zirconia	27
2.6	Callback	28
2.7	Name, Quelle	31
3.1	Diff on Dex level for N1 pattern	41
3.2	Diff on Smali level for N1 pattern	41
3.3	Diff on Java level for N1 pattern (abstracted)	42
3.4	Diff on Dex level for N2 pattern	42
3.5	Diff on Smali level for n2 pattern	42
3.6	Diff on Java level for N2 pattern (abstracted)	42
3.7	Diff on Dex level for N3 pattern	43
3.8	Diff on Smali level for N3 pattern	43
3.9	Diff on Java level for N3 pattern (abstracted)	43
3.10	Diff on Dex level for N4 patch	44
3.11	Diff on Smali level for N4 patch	44
3.12	Diff on Java level for N4 patch (abstracted)	44
3.13	Diff on Dex level for N6 patch	45
3.14	Diff on Smali level for N6 patch	45
3.15	Diff on Java level for N6 patch (abstracted)	45
3.16	Diff on Java level for N7 patch (abstracted)	45
3.17	Diff on Dex level for Amazon patch	46
3.18	Diff on Smali level for Amazon patch	46
3.19	Diff on Java level for Amazon patch (abstracted)	46
3.20	Diff on Dex level for Samsung patch	47
3.21	Diff on Smali level for Samsung patch	47
3.22	Diff on Java level for Samsung patch (abstracted)	47
4.1	asd[16]	51
4.2	Partial Listing	52

List of Code Snippets

4.3	Partial Listing	53
4.4	Partial Listing	55
4.5	Partial Listing	56

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] P. Bernhard. "A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks." Master's Thesis. Technische Universität München, Fakultät für Informatik, Aug. 2015.
- [12] Blackmart. *Blackmart Alpha*. URL: <http://www.blackmart.us/> (visited on 01/20/2016).

- [13] 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).
- [14] 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).
- [15] 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).
- [16] ChelpuS. *Lucky Patcher*. URL: <http://lucky-patcher.netbew.com/> (visited on 01/09/2016).
- [17] 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).
- [18] 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).
- [19] CrackAPK. *Android APK Cracked*. URL: <http://www.crackapk.com/> (visited on 01/20/2016).
- [20] A. Developers. *Licensing Reference*. URL: <https://developer.android.com/google/play/licensing/licensing-reference.html> (visited on 01/21/2016).
- [21] 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).
- [22] D. Ehringer. *The Dalvik Virtual Machine Architecture*. Mar. 2010.
- [23] 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).
- [24] Google Developers. *AdMob for Android*. URL: <https://developers.google.com/admob/android/quick-start> (visited on 01/26/2016).
- [25] Google Play. *Google Play*. URL: <https://play.google.com/store?hl=de> (visited on 01/26/2016).

- [26] GuardSquare. *DexGuard - The strongest Android obfuscator, protector, and optimizer*. URL: <https://www.guardsquare.com/dexguard> (visited on 01/22/2016).
- [27] 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.
- [28] IDC Research, Inc. *Smartphone OS Market Share*. URL: <http://www.idc.com/prodserv/smartphone-os-market-share.jsp> (visited on 01/16/2016).
- [29] T. Johns. *Securing Android LVL Applications*. URL: <http://android-developers.blogspot.de/2010/09/securing-android-lvl-applications.html> (visited on 01/18/2016).
- [30] 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).
- [31] 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).
- [32] A. Kovacheva. "Efficient Code Obfuscation for Android." Master's Thesis. Université de Luxembourg, Faculty of Science, Technology and Communication, Aug. 2013.
- [33] 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-groesser-als-bei-apple/> (visited on 01/16/2016).
- [34] J. Levin. *Android Security - New threats, New Capabilities*. URL: <http://newandroidbook.com/files/Andevcon-Sec.pdf> and (visited on 01/18/2016).
- [35] J. Levin. *Dalvik and ART*. Dec. 2015.
- [36] M. Liersch. *Android Piracy*. URL: <https://www.youtube.com/watch?v=TNnccRimhsI> (visited on 01/22/2016).
- [37] J. F. R. Markus F.X.J. Oberhumer László Molnár. *Ultimate packer for eXecutables*. URL: <http://upx.sourceforge.net/> (visited on 01/22/2016).
- [38] S. Morrow. *Rooting Explained + Top 5 Benefits Of Rooting Your Android Phone*. URL: <http://www.androidpolice.com/2010/04/15/rooting-explained-top-5-benefits-of-rooting-your-android-phone/> (visited on 01/18/2016).

- [39] M.-N. Muntean. "Improving License Verification in Android." Master's Thesis. Technische Universität München, Fakultät für Informatik, May 2014.
- [40] R. Nigam. *Android Packers: Separating from the pack*. URL: <http://www.fortiguard.com/uploads/general/Area41Public.pdf> (visited on 01/22/2016).
- [41] 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).
- [42] R. Price. *DexFile*. URL: <http://www.businessinsider.com/android-app-profitability-v-ios-2015-1?IR=T> (visited on 01/16/2016).
- [43] 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).
- [44] Samsung. *How to protect your app from illegal copy using Samsung Application License Management (Zirconia)*. URL: <http://developer.samsung.com/technical-doc/view.do?v=T0000000062L> (visited on 01/19/2016).
- [45] P. Schulz. "Code Protection in Android." Lab Course. Friedrich-Wilhelms-Universität Bonn, Institute of Computer Science, July 2012.
- [46] M. T. Serrafiero. *Piracy Testimonies, Causes and Prevention*. URL: <http://www.xda-developers.com/piracy-testimonies-causes-and-prevention/> (visited on 01/16/2016).
- [47] ST life.augmented. *Allatori Java Obfuscator*. URL: <http://www.st.com/web/catalog/mmc/FM143/SC1282/PF259413> (visited on 01/24/2016).
- [48] stack overflow. *Posts containing 'Android, Piracy'*. URL: <http://stackoverflow.com/search?q=android+piracy> (visited on 01/26/2016).
- [49] stack overflow. *Posts containing 'Lucky Patcher'*. URL: <http://stackoverflow.com/search?q=lucky+patcher> (visited on 01/26/2016).
- [50] 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).
- [51] 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).

- [52] 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).
- [53] T. Strazzere. *APKfuscator*. URL: <https://github.com/strazzere/APKfuscator> (visited on 01/21/2016).
- [54] T. Strazzere. *dehoser*. URL: <https://github.com/strazzere/dehoser> (visited on 01/21/2016).
- [55] T. Strazzere. *Practicing Safe Dex*. URL: <https://www.youtube.com/watch?v=Rv8DfXNYn0I> (visited on 01/21/2016).
- [56] T. Strazzere. *Practicing Safe Dex*. URL: <http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf> (visited on 01/21/2016).
- [57] TeamSpeak Systems GmbH. *TeamSpeak 3*. URL: <https://play.google.com/store/apps/details?id=com.teamspeak.ts3client&hl=de> (visited on 01/20/2016).
- [58] The University of North Carolina Greensboro. *Software Licensing*. URL: <http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/> (visited on 01/16/2016).
- [59] J. S. Tim Strazzare. *Android Hacker Protection Level 0*. URL: <https://www.youtube.com/watch?v=6vFcEJ2jg0w> (visited on 01/22/2016).
- [60] J. Underwood. *Today Calendar's Piracy Rate*. URL: <https://plus.google.com/+JackUnderwood/posts/jWs84EPNyNS> (visited on 01/16/2016).
- [61] 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.
- [62] R. Yu. *Android packer - facing the challenges, building solutions*. URL: <http://www.allatori.com/> (visited on 01/22/2016).