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

Glossary

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

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

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

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

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

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

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

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

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

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

Acronyms

.dex Dalvik EXecutable file.

.jar Java Archive.

.odex Optimized Dalvik EXecutable file.

ADB Android Debug Bridge.

ADT Android Developer Tools.

AOT Ahead-Of-Time.

APK Application Programming Interface.

APK Android Application Package.

ART Android RunTime.

DRM Digital Rights Management.

DVM Dalvik Virtual Machine.

ELF Extensible Linking Format.

GC Garbage Collection.

IP Intellectual Property.

JIT Just-In-Time.

JNI Java Native Interface.

JVM Java Virtual Machine.

LLVM Low Level Virtual Machine.

LVL License Verification Library.

NDK Native Development Kit.

OS Operating System.

SDK Software Development Kit.

TEE Trusted Execution Environment.

VM Virtual Machine.

1 Introduction

1.1 Licensing

Software Licensing is the legally binding agreement between two parties regarding the purchase, installation and use of software according to its terms of use. It defines the rights of the licensor and the licensee. 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 of usage for the user and prevents him from illicit usage [73].

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

The problem is that these mechanisms do not offer absolute security and pirates always try to circumvent them. This results in an everlasting race of arms between software creators and software thieves [74].

1.2 Motivation

Licensing is present in Android, with a market share of almost 82.8% in Q2 of 2015, as wel [46]l. According to Google, this translates to over 1.4 billion active devices in the last 30 days in September 2015 [30]. This giant number of Android devices is powered by Google Play [44], 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 [77]. 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 [52].
The growth has many advantages. Some time ago developers only considered iOS

as a profitable platform and thus most applications were developed for Apple's Operating System (OS). Now, with Android's overwhelming market share they focus heavily Android [62]. But there are downsides as well. The expanding market for Android, offering many high quality applications, also draws the attention of software pirates. Crackers do not only bypass license mechanisms and offer applications for free. Redirecting cash flows or distributing malware using plagiates is an lucrative business model as well. Android developers are aware of the situation [80] and express their need to protect their IP on platforms like xda-developers [68] or stackoverflow [71]. Many of the developers having problems with the license verification mechanism name *Lucky Patcher* as one of their biggest problems [72].

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

2 Foundation

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

2.1 Software Piracy

According to Apple, 11 billion Dollars are lost each year due to piracy. Software piracy is defined as unauthorized reproduction, distribution and selling of software [25]. It includes the infringement of the terms of use of software by an individual as well as commercial resale of illegal software. Piracy is an issue on all platforms and is considered theft.

2.1.1 Developers

Especially for software developers, piracy is a problem. At first glance, the problem seems to only be theft, where somebody steals a developer's IP and redistributes it without the developer's involvement which results in a loss of revenue. People are then downloading an application either for free or pay the pirate and thus do not generate revenue for the originator.
At second glance, the problems are even more complex. Income for the developer is not only lost when the user is not paying for the software, the pirate can also influence the follow up revenue by modifying the application itself. There are two main types of revenue not generated by the purchase of the application. The first type is the in-app purchase. They are a popular source of income for so called freemium applications or lite versions of apps. In case of the the freemium app, the download is for free and and includes all features. In-app purchases, e.g. in-app currency, enable the user to proceed faster or to buy cosmetic modifications. The lite version application is a little bit different. The download is free as well but the application comes with a restricted feature set or limited time of use. In order to take advantage of the unlocked feature set the user can buy the full license via in-app purchase. Apps can include a mix and various degrees of theses variants. Pirates can disable the payments for the

features, enabling users to receive the in-app purchase for free and thus no earnings are generated for the developer.
The second type of follow up revenue is generated by showing in-app Ads. When this feature is included, advertisements are shown inside the application and the developer is paid by the amount of ads seen and clicked by the user. The Ad Unit ID [43] 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 redirected to the pirate.

Beside monetary issues, additional problems arise when the app is moved to a black market store or website and distributed without the environment of an official app store. This results in the loss of control over the application for the developer. It means the developer can no longer provide the users support or updates for the application to fix crashes caused by malfunctions or security issues. Users which do not know that they are using a pirated version will connect the unsatisfying behaviour to the developer. This can result in the loss of future revenues which are not even connected to this application. In addition to the loss over the application this can cause unpredictable scenarios. Since the developer cannot monitor the growth of the application over tools which are included in the marketplace of choice, he can face unpredictable high traffic. This can stress the server because they were not scaled to the growth. As revenue from the application is stolen, there may not be money for upgrades necessary by legal and illegal use [55].

Developers make a living of their applications. When they do not earn money, because the revenue stream is redirected, because their IP is stolen and commercialized by someone else, or if they even lose money due to uncovered maintaining costs, they cannot continue with developing and their skills are lost.

2.1.2 Users

The loss of developers in the ecosystem is bad for user, but they can also be harmed by piracy. Users use pirated applications because they seem to be free of charge, but it comes at a price. The application might be altered in different ways, e.g. malware may be included or personal data stolen by abusing changed permissions. The user will not notice it right away since these "features" often happen in the background without their knowledge. The application may for instance start using an expensive service, like premium SMS, or upload the contacts to the internet. Even if there is no malicious content implemented, the application can suffer from bad stability due to manipulated code, like removed license verification, and missing updates when related from an unofficial source. In general, the risk is very high that pirated software has a worse user experience than the original. Pirated software should not be installed since it cannot be ensured without deep inspection that the application is doing what it is supposed to do. [29][55]

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 [39]. 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 black market applications, as Blackmart [27], and websites for cracked Android Application Package (APK), such as crackApk [37]. Black market providers claim to be user friendly because they offer older versions of applications. Their catalog includes premium apps, which are not free in the Play Store and include license verification mechanisms, as well. This is only possible when the license mechanism is cracked [24]. They practice professional theft and puts users in danger (see section 2.1.2).
Today Calendar Pro is an example for the dimensions piracy can reach for a single application. The developer stated in a Google+ post that the piracy rate of the application is as high as 85% at the given day [68] [80].
Since license mechanisms are no obstacle for pirates, some developers do not implement any copy protection at all since it is cracked within days [48]. Android applications are at an especially high risk for piracy because bytecode in general is an easy target to reverse engineer as shown in the further proceeding.

2.2 Android

Android is an open source mobile OS launched in 2007 and today mainly maintained and developed by Google. It is based on the Linux kernel and mainly targets touch screen devices such as mobile devices or wearables. The system is designed to run efficiently on battery powered devices with limited hardware and computational capacity. Android's main hardware platform is the ARM architecture, known for their low power consumption, are mainly used in this scenario. The following will give an overview over the architecture of Android and a deeper insight in the runtime system powering Android. The architecture of the software stack of Android can be seen in figure 2.1.

The basis of the system is its kernel. It is responsible for power and memory management and controls the device drivers.

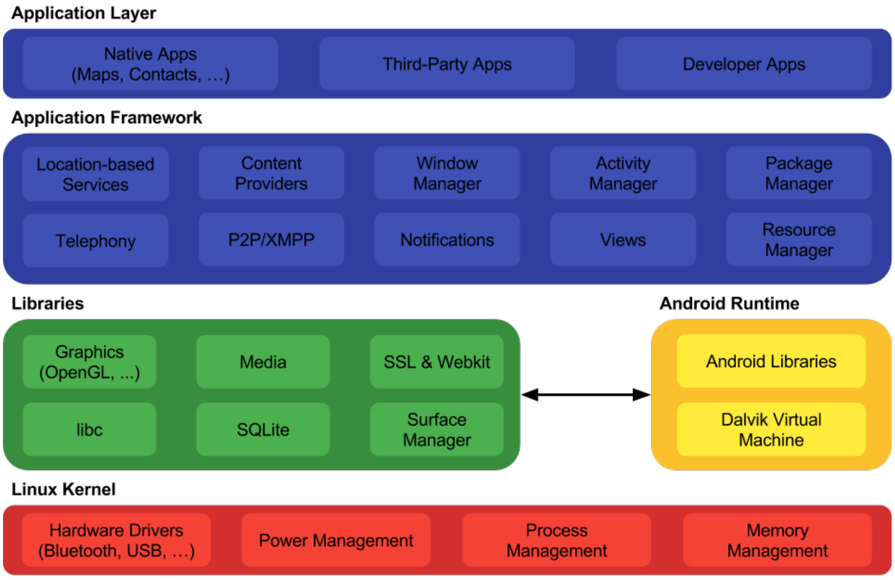

Figure 2.1: Android's architecture [59]

The layer above the kernel contains the Android Runtime, which will be explained in detail in section 2.2.5, as well as the native libraries of the system. Usually Android libraries are written in Java except the ones which are resource and time critical. They are written in C or C++ in order to boost performance and allow low level interaction between applications and the kernel by using the Java Native Interface (JNI). Examples for native libraries are OpenGL, multimedia playback or the SQLite database.
On top of the libraries and the runtime lies the application framework. This layer provides generic functionality as notification support to applications over Android's Application Programming Interface (API).
The top layer enables the installation and execution of applications.
Using these layers and abstraction enables Android to be run on a wide range of devices as well allows software to execute standard Unix commands using the kernel.

2.2.1 Android Application Package (APK)

Android applications are distributed and installed using the APK file format. They can either be obtained from an application store, like Google Play, or downloaded and installed, manually or by using Android Debug Bridge (ADB), from any other source.

The APK format is based on the ZIP file archive format and contains the code and resources of the application. The build process of APK contains several steps which are visualized in figure 2.2.

Since Android applications are usually written in Java, the start is similar to the Java

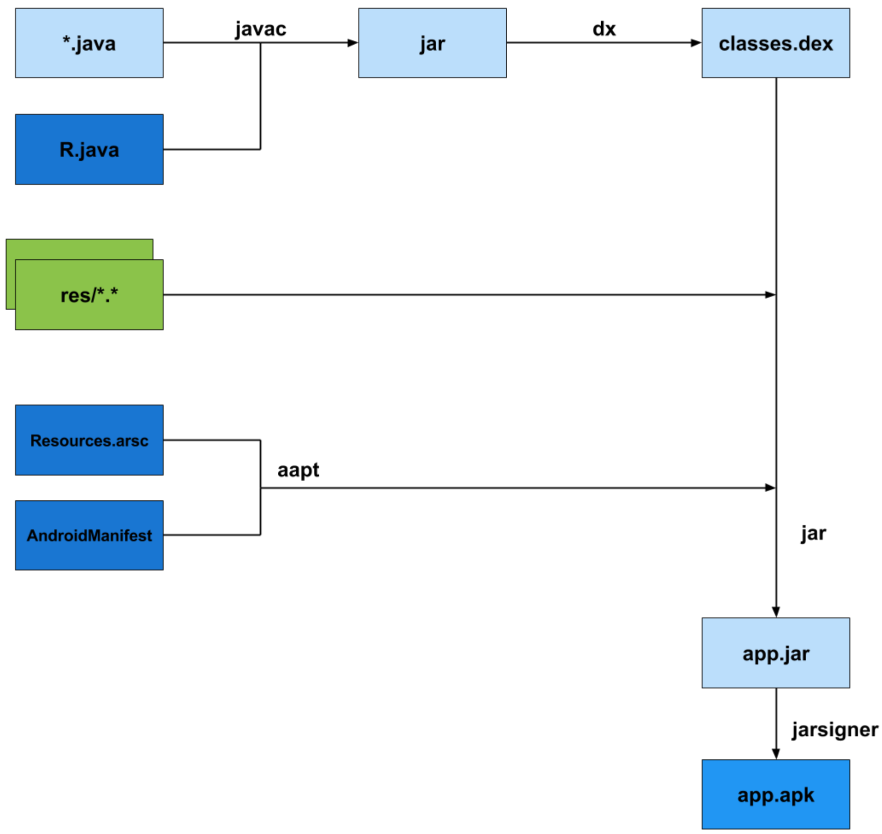

Figure 2.2: APK build process [54]

program build process. The Java source code is compiled into .class file by the Java Compiler javac. Each .class files contains the Java bytecode of the corresponding Java class. As an additional option in the compilation a Java obfuscator can be applied (see section 4.1.3). In the end of the Java compilation process, the .class files are packed into a Java Archive (.jar) file.
Since Android is using the Dalvik Virtual Machine (DVM), which will be described in section 2.2.4, the Java bytecode has to be converted to Dalvik bytecode. The Android Software Development Kit (SDK) includes the tool dx which is used to convert .class files to a single classes.dex file containing all classes. The Dalvik EXecutable (.dex) format will be described in 2.2.2. Additional obfuscation techniques can be applied to protect the .dex code [38].
Now the three main parts of the APK are available:

- classes.dex, contains the bytecode
- resources files (*res/*.**), contains static content like images, layouts and native code
- resources.arsc and AndroidManifest.xml, contain compiled resources respectively essential information like needed permissions

These parts are combined by the ApkBuilder into one archive file.
Finally, jarsigner adds the developer's signature to this package which does not improve security of the application itself, but identifies the developer and thus supports updates.

The structure of a final application file has at least the following content seen in figure fig:apkfolder. The AndroidManfiest.xml and the classes.dex, which have been

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

Figure 2.3: APK folder structure

covered already. The META-INF folder, which is inherited from Java and used to store package and extension configuration data and other [60]. While the static resources, like drawables and layouts, are in the res folder, the resources.arsc contains the compiled resources. In case the application implements native code, it is stored in the libs folder, split by the different processor types, like armeabi-v7a for ARM or x86 for Intel processors. [50] [40]

2.2.2 Dalvik Executionable File Format

As explained in subsection 2.2.1, Android applications deliver their code in .dex bytecode and are executed by the DVM. The dex file format is compiled from Java bytecode. It is similar to Java bytecode except some differences. The biggest difference is the concept how the code is executed. While the Java Virtual Machine (VM) is stack-based the DVM is register-based, this circumstances have influence on the code. In addition the Dalvik bytecode is more suited to run on the ARM architecture since it supports direct mapping from dex registers to the registers of the ARM processor. Registers in .dex bytecode are 32bits wide and store values such as integers or float values. In case there are 64bit values, adjacent registers are used to store it. The Java bytecode is actually more compact since it uses 8bit constants while .dex bytecode has instructions of 16bit multiples. The .dex bytecode supports 218 valid opcodes which have a source-destination ordering for its arguments. The arguments of the instruction refer to indexes in the pools which can be seen in figure 2.4.

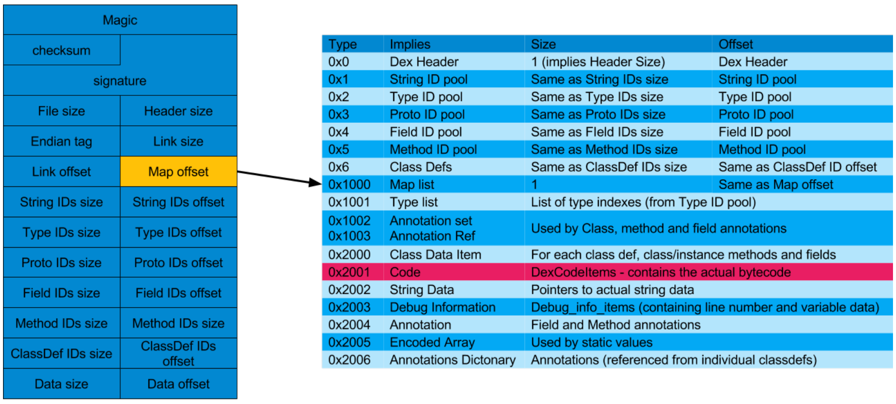

Figure 2.4: .dex file format [54]

In order to create the dex file, dx is used on the .jar file. Dx compiles the Java bytecode to .dex byte code and sorts string, type and method from the heterogenous pool of each class into global pools. This results in figure 2.4 and figure 2.5.

When merging the resource pools, dublicates are removed. This is most effective for string and results in a decrease of the memory footprint by up to 44% lower in contrast to the .jar. The result is that the .dex file has significant more references than the .jar file. The .dex file is stored as classes.[40]

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

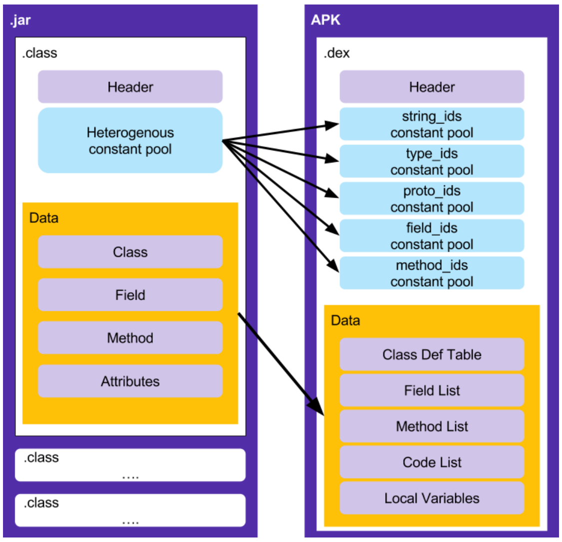

Figure 2.5: .jar to APK transformation [28]

lation to Java. Since the bytecode is, in contrast to other architectures, pretty simple to understand as well as protection is applied rarely since it has to be done by the developer itself, it is an easy target for reverse engineering.

2.2.3 Installing an APK

Now that the format of the APK and .dex file format are explained, the application can be installed. Before running an application, two steps are applied on the APK. The first step is primary about the verification of the application while the second step is the bytecode optimization and, in case of Android RunTime (ART), the code compilation. Before installing an application it is checked for a legitimate signature as well as correct classes.dex structure (see figure 2.6). In case this cannot be verified, the installation will be rejected by the OS. The second step is the optimization. In case the device is still running the DVM, the .odex verison of the classes.dex is generated. This is possible because .dex files can be optimized in order to achieve the best performance for a given device architecture. This is due to the the high diversity of Android running hardware and their different processors. In the process the classes.dex file is taken from the archive and put as .odex file into the Dalvik cache. This is done once and from then on the execution is done by using the .odex file. This preprocessed version of the application has an improved startup time. [50] The current versions of Android run on the ART. For this runtime the second step is more complex since the bytecode has to be compiled an additional time. This will be explained closer in section 2.2.5.

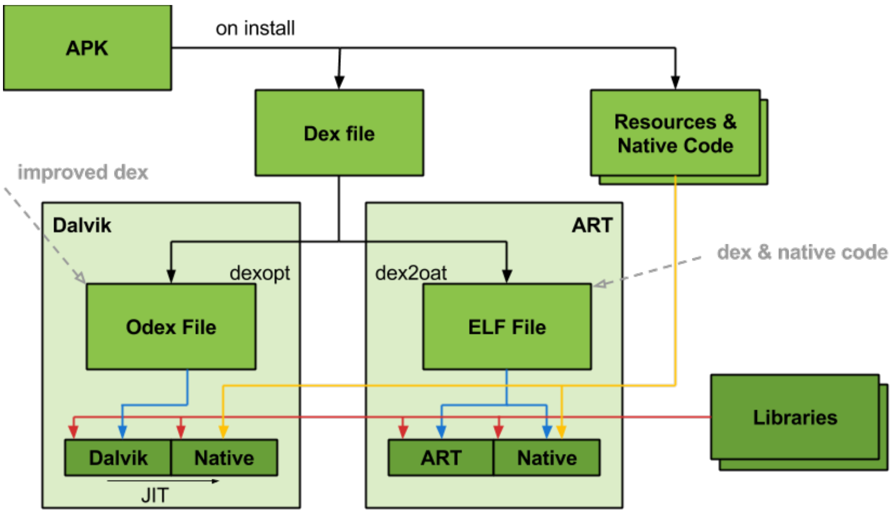

Figure 2.6: Installing an APK on a device [23]

When the application is run later, Android is creating an sandboxed environment for each application. This is achieved by Android assigning each process a seperate user ID on install to ensure that each application is isolated from the others and has no access on resources except its own. [13]

2.2.4 Dalvik Virtual Machine

The original VM powering Android is the DVM, which was created by Dan Bornstein and named after an Iclandic town. It was introduced along with Android in 2008 [21]. In contrast to a stationary computer a mobile device have a lot of constraints. Since they are powered by battery the processing power and RAM are limited to fit power consumtion. In addition to these hardware limitations Andriod has some additional requirements, like no swap for the RAM, the need to run on a diverse set of devices and in a sandboxed application runtime . In order to deliver best performance and run efficiently it has to be designed according to these requirements. The DVM is a customized and optimized version of the Java Virtual Machine (JVM) abd based on Apache Harmony. Even though it is based on Java, it is not fully J2SE or J2ME compatible since it uses 16bit opcodes and register-based architecture in contrast to the standard JVM being stack-based and using 8bit opcodes. The advantage of register-based architecture is that it need less instructions for execution than stack-based architecture which results in less CPU cycles and thus less power consumption

which is important for a battery driven device. The downside of this architecture is the fact that it has an approximatly 25% larger codebase for the same application and negligible larger fetching times [40]. In addition to the lower level changes the DVM is optimized for memory sharing, it stores references bitmaps seperately from objects and optimizes application startup by using zygotes [54].
The last big change made to the DVM was the introduction of Just-In-Time (JIT) in Android version 2.2 *Froyo*.

2.2.5 Android Runtime

<!– benötigt noch feinschliff ->

In Android version 4.4 *Kitkat* Google introduced ART which was optional and only available as a preview through the developer options. Like DVM, ART executes the .dex fileformat und .dex bytecode specification [9]. With the release of verison 5.0 *Lollipop* ART it became the runtime of choice since DVM had some major flaws. Throughout the Android 6.0 *Marshmallow* previews it was constantly evolving and sometimes breaking with older versions at the cost of almost no documentation.
ART is designed to address the shortcomings of the DVM. Maintaining an VM is expensive, having an interpreter and JIT is not as efficient of native code and doing JIT each time the application is executed is wasteful. This and maintaining background threads require significantly more CPU cycles which can be directly translated to slower performance and increased battery usage. In addition the Dalvik Garbage Collection (GC) frequently causes hands and jitters and DVM is only supporting 32bit. With ART Android is following iOS into the 64bit world but this is not the only advantage over the DVM. Improvements in the VM make the maintenance less expensive and the compilation is changed from JIT to Ahead-Of-Time (AOT) as well as the overhead cycles have been reduced. The GC is also non-blocking now and and can run parallel in fore- and background.
The mean idea of ART and AOT is to compile the application to one of two types, either native code or Low Level Virtual Machine (LLVM) code. Each of the types has its purpose and advantage. The native code offers an improved execution performance while the LLVM code offers protability. In practice the preference is to compile to native since adding LLVM bitcode adds another layer of complexity to ART.
Different from DVM is the fact that ART uses not one but two file formats. Similar to the zygote of DVM, ART offers an image of pre-initialized classes and and related object at run time, the boot.art file. It is mapped to the memory upon zygote startup to provide improved application starting time [10]. The second file format is

Art itself: art uses not one but two file formats art - only one file, boot.art, in

/syste/framework/architecture (arm,...) oat - master file, boot.oat, in /syste/framework/architecture (arm,...) - odex files no longer optimized dex but oat, alongeside apk for system apps/frameworks, /data/dalvik-cache for 3rd party apps, still uses odex extension, now file format is elf/oat
art files is a proprietary format, poorly documented, changed format internally repeatedly art file maps in memory right before oat, which links with it contains pre-initilited classes, objects and support structures
ART/OAT files are created (on device or on host) by dex2oat art still optimizes dex but uses dex2oat instead, odex files are actually oat files (elf shared objects WAS IST DAS), actual dex payload buried deep inside command line saved inside oat file's key value store

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

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

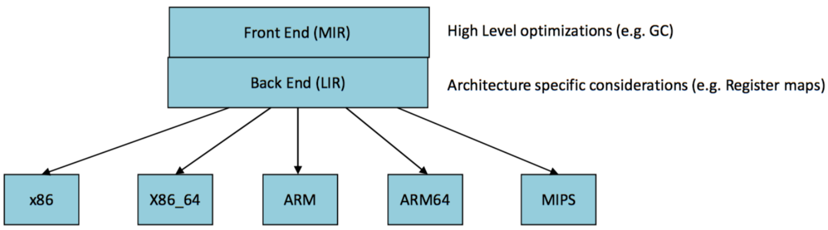

Figure 2.7: artarch

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 potentilly hundres of classes

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

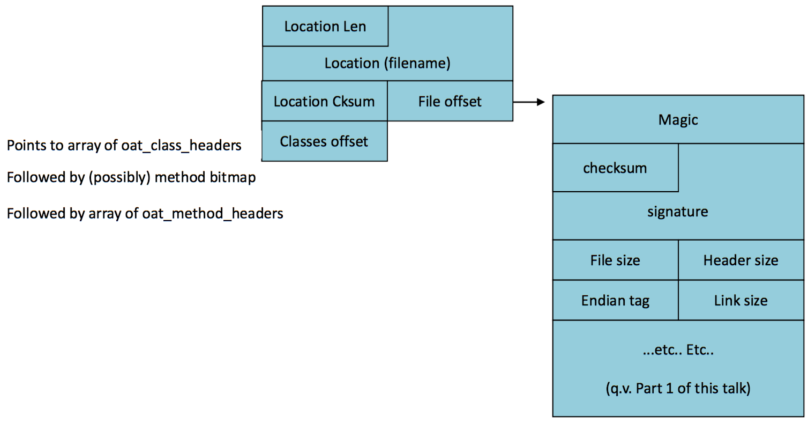

Figure 2.8: oatdex

dex code doesnt use them generated code isnt always that efficient, not on same par as an optimizing antive code compiler, likely to improve with llvm optimizations overall code (determined by Mir optimizations) flow is same garbage collection, register maps, likewase same cavears: not all methods guaranteed to be compiled, reversing can be quite a pain

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

2.2.6 Copy Protection and Root

Now that the application is installed and ready to run. In order to prevent unauthorized usage of the app copy protection is applied. After downloading APK from an application store it is moved to /data/app on the phone upon installation. The user has no rights to access this folder and thus not copy the application. This mechanism has a major flow since when a single user can get hold of the APK and redistribute it, copy protection for the application is circumvented. This was effective in the early days of Android when rooting was not easily facilitated.
Rooting or *getting root* is the process of modifying the operation system's software that shipped with a device in order to get complete control over it. The name *root* comes

from the Linux OS world and is the user with the most privileges. This allows the user to overcome limitations by carriers and manufacturers, like removing pre-installed applications, extend system functionality or to upgrade to custom versions of Android. Manufacturers and carriers do not aprove rooting but since the access is usually gained by exploiting vulnerabilities in the system's code or device drivers, they cannot prevent it. Vulnerabilities are quite common and documented [1]. Today it is easy, even for non-techies, to gain root. There are videos and tutorials on the internet, even tools to automate the process, like Wugfresh's Rootkits [81], are available. Rooting is usually bundles with installing a progremm called *su* which manages the root access for applications which request it. Rooting a phone is not without risk since installing bad files can result in the so called *bricking* meaning the phone is nonfunctional since the software cannot be executed anymore.[57]
Even though rooting brings a lot of improvements to the table it is bad for the copy protection making it non-existent. For this reason new ways of protecting applications have to be invented.

TODO: 1) Root Exploits Man koennte sagen die Wurzel allen uebels - im Sinne von Kopierschutz. Daher bitte alle Root-Exploits raussuchen, die es in den letzten Jahren so gab und auf der benannten Timeline darstellen (s.u.), damit wir auch einen Rueckschluss ziehen koennen, wie oft es Root-Exploits gibt. Weitere Exploits haben Kommilitonen von Dir auch mal im AP erstellt. Bitte alles zusammentragen und in eine Grafik ueberfuehren.

Von Interesse ist auch, welche Geraete eigentlich offiziell gerootet werden koennen. Wie sehen das die Hersteller? Zunehmend sieht man Secure Boot und dergleichen?

2.3 License Verification Libraries

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

attractive and bind developers with low piracy rates. Since the initiative is coming from the stores, the copy protection methods are included into application itself. [58]

2.3.1 Google's License Verification Library (LVL)

In order to tackle the copy protection problematic and to give the developer community a possibility to fight piracy, Google introduced the License Verification Library (LVL) at 07/27/2010 [34]. It is simple and free of charges. The documentation can be found on the Android developers website. [17]

Functional Principle

Google's approach is a network service which allows to query the trusted Google Play license server in order to determine whether the user has a valid license. The library is provided by Google and the exact functionality can be studied by the developer since it is delivered in source code.[58]
It has to be manually integrated into the application by the developer and allows a simple checking with Google and has a callback for the reply. The developer can decide when and how often the application should check its license. Upon initiation, the library connects to the Google Play Service which manages the connection between the device and the license server. It sends a request for a license check on the server to determine the validity of the license, whether the application was bought by the user. This is similar to Digital Rights Management (DRM). Adding the library to the application does not alter the function of the application, it just addes a call. While the library takes care of the complicated process, like the networking and webservice, and returns the response of Google Play as soon as it arrives, the developer only has handle the callback. The developer is in full control what should happen with the response and whether access should be granted. Figure 2.9 gives an overview how the parts are connecnted. [49][17]
In order to start the validation process with the Google Server und to determine the license status of the application, additional information has to be provided. The user has to initiate the call with the package name of the application, a nonce, which has to be included in every repsonse to ensure integrity, as well as the callback for asynchronous handling. The Google Play Service then adds the primary Google account the application is executed with, the IMSI of the device On the server Google checks whether the user has purchased the application and sends the corresponding response to the Google Play Service which passes it to the application. [17]
The security of the response is very important. It is achieved by public/private key encryption. Each published application in the Play Store has a pair of keys of which the

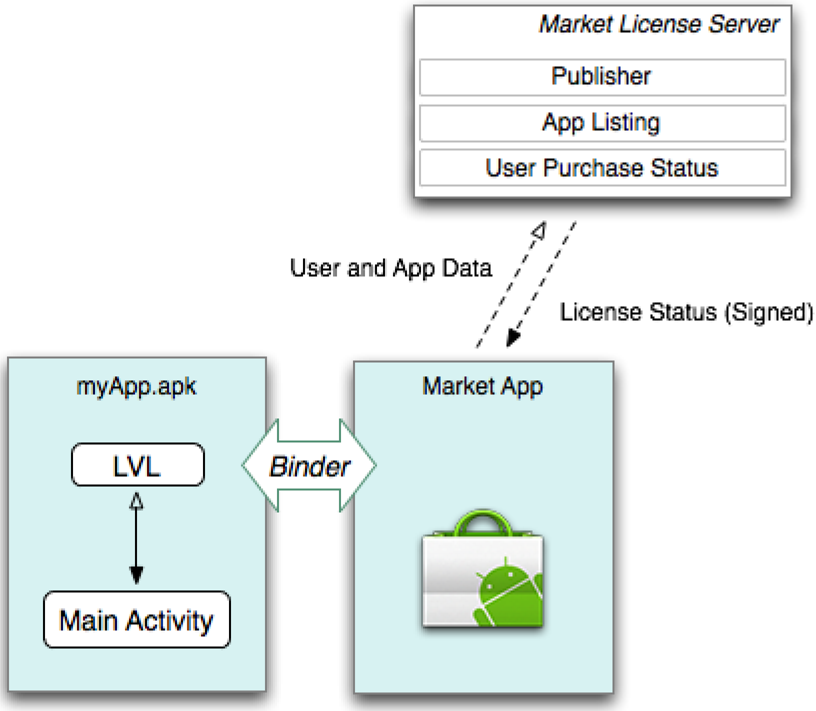

Figure 2.9: Google's implementation of license checking [17]

developer has to integrate the public key, which is visible to him, into the application in order to decrypt the communication which is encrypted by the server with the private key. This way it the origin of the response is ensured as well as tampering detected. [58][17]
This results in some restrictions for the application to run. First of all, access to the internet is necessary as well since a connection to the Google Server has to be established. In case this is not possible, an internal error will be triggered which results in the license not being verified. Second of all, the Google Play Service has to be installed. It comes preinstalled with the Google Apps which includes the Google Play store. On the one hand, this is a prerequisite to legally aquire an application with the LVL. On the other hand, in case applications, with LVL implemented, cannot bind to the Google Play Service and thus cannot verify the license status. When the server is finally reached and the license is verified, it can be stored on the device and used in future requests in case no internet connection is available. [11][17]
In summary this means the Google Play Service as well as a one time internet connection have to be present on the first startup in order to make the license check and, in case of

the right answer, run the application. It replaces the old copy protection, which is no longer supported, with a secure mechanism to control access to the application. This license verification model can be enforced on all devices which have access to Google Play. [11] [17]

Implementation of the License Verification Library

The provided library of Google can be implemented in only a few steps. First of all, a Google Publisher Account for Google Play is needed. It enables the developer to publish applications on Google Play and take advantage of the LVL. As soon as an application is created in the Google Developer Console the public/private key is created. Each key pair is app specific and can be found under "Services & API". It is later on added into the application. [19]
Now that the Google Play prerequisites are set, the LVL has to be extracted from the Android SDK folder and added to the application. As soon as the LVL is part of the application project the licensing permission can be added to the application's manifest (see code snippet 2.1), else the LVL will throw an exception. The last step is to create

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

Code Snippet 2.1: Include permission to check the license in AndroidManifest.xml [11]

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

```
    final String mAndroidId = Settings.Secure.getString(this.getContentResolverSettings.Secure.ANDROID_ID);
    final AESObfuscator mObsfuscator = new AESObfuscator(SALT, getPackageName(), mAndroidId);
    final ServerManagedPolicy serverPolicy = new ServerManagedPolicy(this, mObsfuscator);
    mLicenseCheckerCallback = new MyLicenseCheckerCallback();
    mChecker = new LicenseChecker(this, serverPolicy, BASE64_PUBLIC_KEY);

    mChecker.checkAccess(mLicenseCheckerCallback);
```

Code Snippet 2.2: Setting up the LVL license check call

different information has to be aquired and added to the license request. In order to identify the user the its unique id is aquired. The id is created randomly when the user sets up the device for the first time and remains the same for the lifetime of the user's device [20]. The obfuscator is later applied on the cached license response data to prevent manipulation or reuse by root user. In order to decide what should happen to the reponse data and whether it should be stored, a policy is specified. This can be either one of the provided policies are a custom one. Before making the license check request, a callback has to be definden, which can be seen in code snipped 2.3. The

```

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

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

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

Code Snippet 2.3: LVL license check callback

callback contains actions for the different outcomes of the license check. The scenarios are either applicationError, e.g. when no connection can be established, dontAllow in case license is not valid and allow when the users status is verified. Now the license check can be initiated with the values. [19][11]
The license check can be implemented in any activity and executed as the developer likes.

2.3.2 Amazon DRM (Kiwi)

As an alternative to Google Play, started its own application store in October 2010 with the goal to generate profits from selling apps on Android as well [7]. It was opened to the public on the 03/22/2011 for Android and Fire tablets [8]. The store comes with its own DRM since the Google LVL only works with the Google Play Store. The developer can chose whether it should be enabled in the developer console in their developer console. When reengineering the APK the added code can be found in a package called Kiwi as seen in figure 2.10. [6]

Figure 2.10: Amazon library structure in decompiled application

Functional Principle

The prerequesites are the same as with the LVL, the developer has to have an account on the Amazon platform. Amazon has a different approach of implementing the license verification check. Instead of the developer integrating the license verification library himself, the developer is asked when uploading the application whether it should be integrated (see figure 2.11). According to the description, the library is to *Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.* [6] In order to implement the library the APK is decompiled server sided, the library is added and the package is then signed with a new signature (see subsection 2.2.1). Amazon describes it as following: *As part of the ingestion process Amazon removes your developer signature and applies an Amazon signature. This signature is unique to you, does not change, and is the same for all apps in your account.* [6]

applay amazon DRm to *Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user.* as the description says as the description says, so developer signing the application by the develoepr before submitting is not necessary, amazon decompiles apk, injects drm code, compiles it and signs it with the *Amazon developer* certificate

when uploading the app, user is asked whether

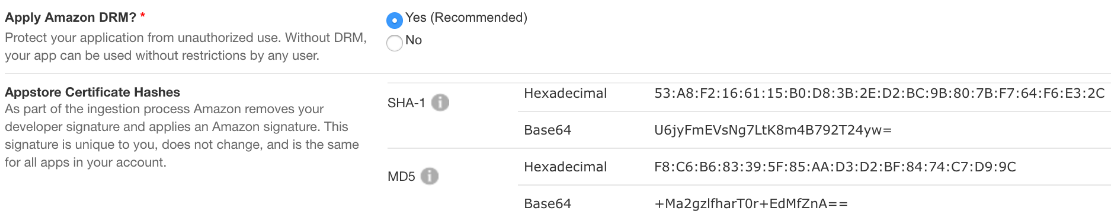

Figure 2.11: Developer preferences in the Amazon developer console [6]

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 on their own, done by amazon packaging tool when submitting can check amazon

DRM (see picture), applay amazon DRm to "Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user." as the description says "As part of the ingestion process Amazon removes your developer signature and applies an Amazon signature. This signature is unique to you, does not change, and is the same for all apps in your account." as the description says, so developer signing the application by the develoepr before submitting is not necessary, amazon decompiles apk, injects drm code, compiles it and signs it with the "amazon developer" certificate [58]

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

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

Implementation of Kiwi

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

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

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

[58]

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

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

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

2.3.3 Samsung DRM (Zirconia)

Samsung as a major player in the smartphone business has also his own app store [35] Galaxy Apps by Samsung, formerly known as Samsung Apps, for devices by Samsung renamed in July 2015 called zirconia for android [66]

Functional Principle

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

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

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

[66]

no store services needed, direct communication with server

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

Implementation of Zirconia

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

4 basics steps (see 2.5) create

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

[66]

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

Code Snippet 2.5: Setting up the Zirconia license check call

First looks great, puts the copy protection inside the app, a from of DRM communicate with server, authorize use of application
does not prevent user from copying/transfering app, but copy useless since the app does run without the correct account

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

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

Code Snippet 2.6: Zirconia license check callback

google die ersten, andere folgen, anfangs problem, dass dadurch nur durch google store geschützt war, grund dafür dass evtl ein programmierer in meinen store kommt

2.3.4 Abstraction

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

but there are problems as well android's content protection are invalid when rooted DRm can be bypassed coupled with dex decompilkation big problem, app can be decompiled, modded and repackaged[53]

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

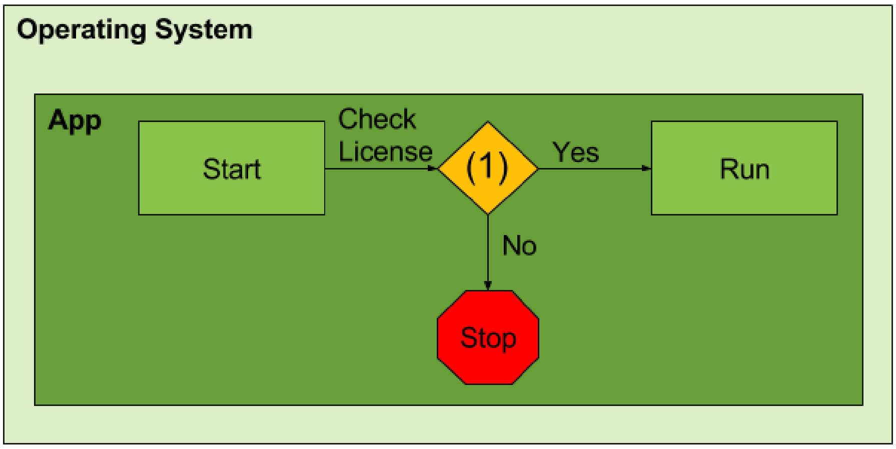

Figure 2.12: Abstraction of the current license verification mechanism. The library is represented by (1)

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. [50]

dex to java .dex and .class are isomorphic dex debug items map dex offsets to java line numbers tools like dex2jar can easily decomile from dex to a jar extremly useful for reverse engineering, even more so useful for malice
flow from dex to java is bidirectional, decompile code back to java, remove annoyances like ads, registration, uncover sensitive data (app logic, secrets), replace certain classes with others (malicuous ones), recompile back to jar, then dex, put cloned/trojaned version of your app on play or other market

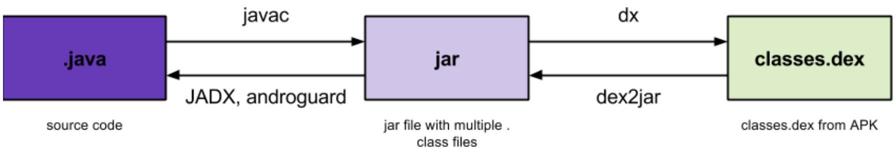

Figure 2.13: Java .class and .dex can be transformed bidirectional [54]

[54]

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

many tools and techniques

gaining information about a program and its implementation details, process aims at enabling an analyst to understand the concrete relation between implementation and functionality, optimal output of such a process would be the original source code of the application, not possible in general
Therefore, it is necessary for such a process to provide on the one hand abstract information about structure and inter-dependencies and on the other hand result in very detailed information like bytecode and mnemonics that allow interpretation of implementation
hoffentlich starting points für investigations
java, e.g. read the program code faster

was ist reengineering? wie funktioniert es? was ist das ziel?
reverse engineering process makes use of a whole range of different analysis methodologies and tools.
only consider static analysis tools

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

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

• It comes as no surprise that .dex and .class are isomorphic • DEX debug items map DEX offsets to Java line numbers • Dex2jar tool can easily “decompile” from .dex back

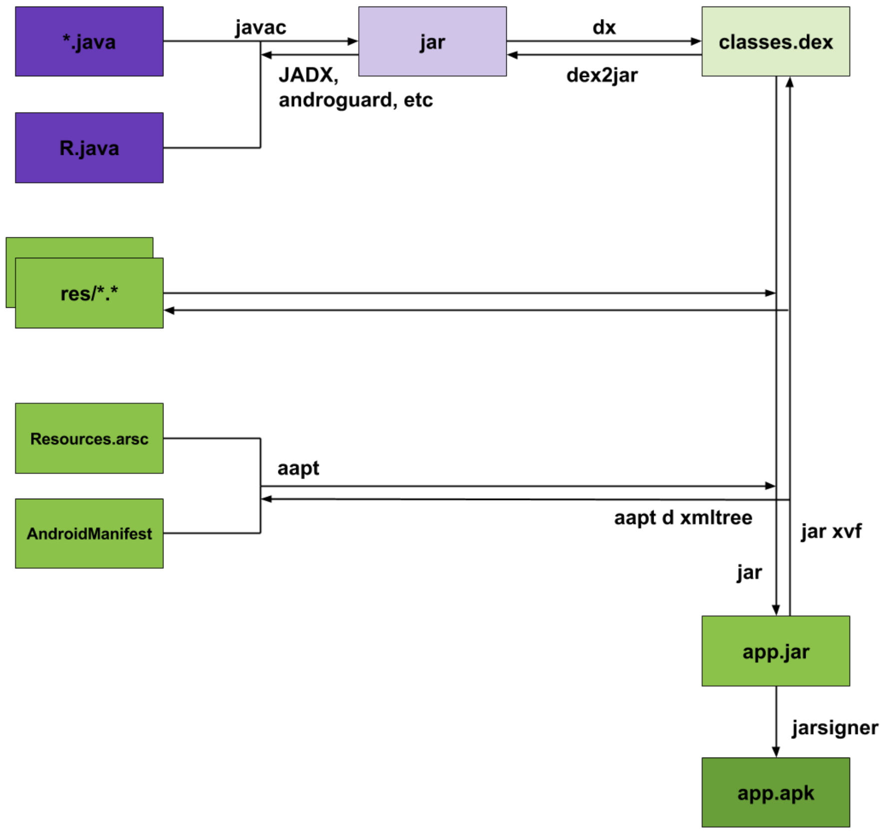

Figure 2.14: Overview of build and reverse engineering of an APK [54]

to a .jar • Extremely useful for reverse engineering – Evenmoresousefulformaliceand-mischief

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

`https://mobilesecuritywiki.com/`

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

main tools

2.4.1 Retrieving an APK

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

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

then find app which should be transfered to computer and use adb pull /data/app/me.neutze.licensetest 1/base.apk to download app into current folder

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

[58]

In the following there will be an example application to generalise the procedure. The application is called L̈icenseTeständ has for our purpose a license verification library included (Amazon, Google or Samsung).
In order to analyse an APK, it has to be pulled from the Android device onto the computer. First the package name of the app has to be found out. This can be done by using the ADB. Entering example 1 returned example 2

dann auf verschiedenen leveln anschauen mit den folgenden tools

2.4.2 dex Analysis

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

nur dex weil die apps im moment so vorliegen

aosp-supplied dexdumo to disassemble dex

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

hexdump to get bytecode

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

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

```
#!/bin/bash
#hexdump dex
unzip lvltest-version.apk -d /tmp/
hexdump -C /tmp/classes.dex >> /dex/version/classes.txt
```

Code Snippet 2.7: Script to extract the .dex byte code from the APK dump, start of the dex file, e.g first 8 byte represent the dex header magic dex.035, version 35[15] LINE | bytecode | ASCII representation

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

Code Snippet 2.8: dex hexdump example

jedes tool:
woher kommt es?
wozu wurde es erfunden?
wer hat es erfunden? quelle
blabla von der seite
wozu benutze ich es?
welches abstrahierungslevel
beispiel
additional features?
WARUM SCHAUEN WIR ES UNS AN?
wo findet man es?
welches level?
vorteil
blabla aus dem internet

2.4.3 Smali Analysis

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

basically jasmin syntax smali, most popular Dalvik bytecode decompilers (used by multiple reverse engineering tools as a base disassembler, amongst which is the also well-known apktool) [50] dex == smali, smali better readable but dex to see how easy change since the translation from java to dex does some optimizations/logik, dex and java do not express the same, but it is how it is in the decompiled code, java is also an abstraction of the actual code, sometimes java also a little confusing since changes happened in dex code and cannot be decompiled to java in a good manner, very messy, it is included for better understanding anyways since humanreadable

stichwort mnemonics, eine seite dex und auf der anderen seite smali, dex bytecode

vs smali, Only a few pieces of information are usually not included like the addresses of instructions
unintuitive representation, deswegen smali mit corresponding mnemonics
mnemonics and vice versa is available due to the bijective mapping
correct startaddress and offset can be challenging. There are two major approaches: linear sweep disassembling and recursive traversal disassembling, The linear sweep algorithm is prone to producing wrong mnemonics e.g. when a assembler inlines data so that instructions and data are interleaved. The recursive traversal algorithm is not prone to this but can be attacked by obfuscation techniques like junkbyte insertion as discussed in section 4.4. So for obfuscation, a valuable attack vector on disassembling is to attack the address finding step of these algorithms

https://github.com/JesusFreke/smali

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

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

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

SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)

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

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

EXAMPLE BESCHREIBEN code snippet 2.10

jedes tool:
woher kommt es?
wozu wurde es erfunden?
wer hat es erfunden? quelle

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

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

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

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

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

Code Snippet 2.10: smali example

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

2.4.4 Java Analysis

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

dex different patterns for mobile Usage, java does not really now, thats why different java decompiler

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

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

androguard

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

An analysis and disassembling tool processing both Dalvik bytecode and optimized bytecode
DAD which is also the fastest due to the fact it is a native decompiler, WAS ist dad? ERKLÄREN? .dex files was performed with DAD, the default disassembler in the Androguard analysis tool, largest successful disassembly ratio
underlying algorithm is recursive traversal
androguard has a large online open-source database with known malware patterns [50]

`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)
SCRIPT (LISTING BENUTZEN UND RICHTIGE APP)

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

Code Snippet 2.11: Script to decompile to Java using androguard

EXAMPLE BESCHREIBEN code snippet 2.12

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

Code Snippet 2.12: Java code example using androguard

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)

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

Code Snippet 2.13: Script to decompile to Java using JADX

EXAMPLE BESCHREIBEN code snippet 2.14

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

Code Snippet 2.14: Java code example using JADX

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

2.4.5 Detect Code Manipulations

This is my real text! Rest might be copied or not be checked!
vergleich gibts guten einblick was geändert wurde und wie es auf dem gegebenem lvl funtkioniert

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

diff is used

```
#!/bin/bash
#dex
diff -Naur /dex/original/ /dex/manipulated/ > dex.diff
#smali
diff -Naur /baksmali/original/ /baksmali/manipulated/ > baksmali.diff
#jadx
diff -Naur /jadx/original/ /androguard/jadx/ > jadx.diff
#androguard
diff -Naur /androguard/original/ /androguard/manipulated/ > andro.diff
```

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

erklärung command [79] -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

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

3 Cracking Android Applications with Lucky Patcher

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

3.1 Lucky Patcher

Lucky Patcher is described as "[...] a great Android tool to remove ads, modify apps permissions, backup and restore apps, bypass premium applications license verification, and more" on the official website [32]. It is written by ChelpuS and currently on version is 6.0.4 (on 02/17/2016). Since applications are stored that the user cannot access them (see subsection 2.2.6), root is necessary to use the application. In addition, busybox, an application which provides standard UNIX tools for Android[75], is required as well.
Lucky Patcher offers the removing of the licensing in premium apps to crack their DRM, to remove in app ads, change and restrict permissions and activities as well as to create modified after applying one of the feature above on the original APKs [32].
The application requires, besides of rooting, no technical knowledge to handle and offers automatic cracking for non professionals. This combination makes it a popular and an effective tool with a high damage potential. [58]
This thesis focuses on how Lucky Patcher is bypassing the license verification mechanism of applications. The goal of circumventing the license check is to make the pirated application to work as it would have been legally acquired in the store. As described in section 2.3, the license verifications are implemented as client-server connection. The app gathers the information, sends it to the server, which checks the verifies the given information, and replies the result to the client. Since the server is not accessible, an man-in-the middle attack or spoofing would be an obvious options to break the license verification mechanism. The execution of these is very difficult since breaking encryption in general is hard to carry out [58]. This is the reason why Lucky Patcher is taking a different path by modifying the application itself. This will be analysed in

detail in the following chapters after explaining the general usage.

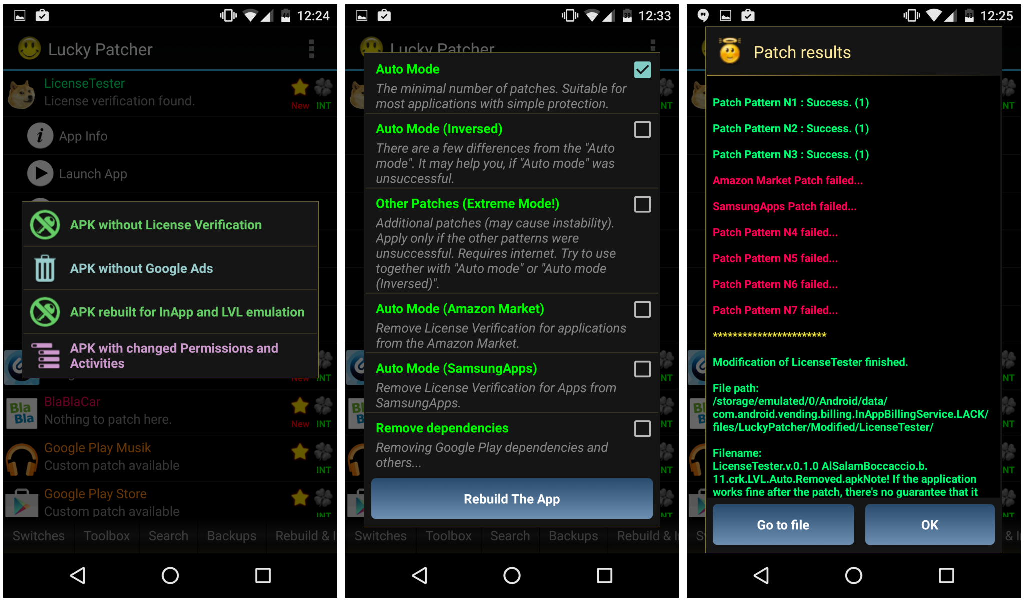

Figure 3.1: c
Left to right: Features offered LuckyPatcher, modes to crack license verification and the result after patching

Using Lucky Patcheris fairly simple. The application can be downloaded from the official website [32] as an APK and installed on the device. On startup, all installed applications are shown in a list. Below their name, an information text about what patches can be applied is shown. When a the application is selected, a submenu offers actions like to get information about the app or to run it to name a few. The action required to modify the application is the menu of patches which offers the features described above and can be seen on figure 3.1 on the left. In this menu there are two choices to achieve the license verification. In case the crack should be applied directly on the device by creating an odex file, the license verification removing should be chosen. This is only possible when root is available on the device. In the other case an modified APK can be created, which includes the modified license verification mechanism. and works on unrooted devices as well since it does not access restricted folders. When the output of choice is selected, different modes can be selected. The description of what they are doing can be seen in figure 3.1 in the middle and is rather short. A closer analysis of how Lucky Patcher is working is done and presented in section 3.3. After choosing the mode of choice the actual patching is done and the

result are shown as seen in figure 3.1 on the right. The different patching patterns will be analysed and explained in section 3.4. Lucky Patcher does not offer a 100% warranty that the result is working and all license verification related restrictions are removed.
For this thesis Lucky Patcher is analysed in two different ways. On the one hand the source code reengineered and the functionality of the reversed code is inspected. On the other hand the cracking tool is applied to different applications and the result was compared to the original in order to see how Lucky Patcher is applying patches.

3.2 Code Analysis

The code analysis was done using Lucky Patcher in version 6.0.4 using the two tools described in subsection 2.4.4. The reversed code was inspected using a text editor like Atom [42].
Before analysing the code, a look is taken at the structure. When loading the folder of the reengineered code into the editor a lot of different code folders can be spotted. These code folders hold the packages of Lucky Patcher itself and used libraries. The libraries can be divided in four categories. The first category contains the Android Support Library v4 with many of it's modules, which Lucky Patcher includes as well as many other applications. It is used for downward compatibility of Android related functions of Lucky Patcher itself. The second category contains the code of Lucky Patcher. It is located in two places. In the package com.chelpus utility functions, like *copy file* and *rename*, are stored. The *application* itself is com.android.vending.billing.InAppBillingService.LACK. It contains the activities and functions which are used for cracking applications. The third category are support libraries required by Lucky Patcher to apply it's cracking mechanisms. This includes libraries like axml [33] for serializing the AndroidManifest.xml from Android binary into an ASCII formatted xml file to make it readable and zip4j [56], a Java library to handle ZIP files, to name a few. The fourth and last category is an modified billing and license library, which can be applied in combination to use a proxy to redirect inapp billing and licensing calls, and a package containing the already cracked LVL which will be explained in section 3.4. In addition to these categories, the asset folder stores different predefined custom patches which can be applied applications [58].
This is very hard since there is no folder structure sorting the classes in activities or models. The patchActivity in the package com.android.vending.billing.InAppBillingService.LACK can be identified as launcher method when looking into the AndroidManifest.xml file. The folder embracing the activity has a flat hierarchy and is not grouped into folders by activities or models. In the launcherActivity the listAppsFragment is created which implements most of the app logic in over 17.000 lines of code. The additional files are

models and classes implementing functions, e.g. CorePatch for writing the output file. This makes it difficult to get a clear understanding of how exactly Lucky Patcher is patching the attacked application. Another problem is the obfuscation of strings and methods which makes it impossible to get their purpose just from the name. At some points it is even impossible for the decompiler tool to disassemble to Java code and thus it returns just the instruction dump.

Lucky Patcher has already been target of analysis, e.g. in Marius-Nicolae Muntean's Master's thesis "Improving License Verification in Android" [58] thus there are functions to look for. When loading the applications, each package is checked for it's permissions which are extracted from the application's AndroidManifest.xml. When the application with the check is selected for cracking, the classes.dex is extracted from the APK. In the next step Lucky Patcher applies the manipulation of the .dex file on binary level. The altered file can either be output as an APK or put, depending on the runtime, either as .odex in the Dalvik Cache at /data/dalvik-cache/ or directly injected into the corresponding .art file. After changing the classes.dex, the header's checksum hash and signature have to be updated to reflect the changes and to be valid. [58]

3.3 Analysis of Patched Applications

In addition to analysing the reverse engineered source code, an analysis of patched applications is done. This is done by first patching an application and creating a modified apk as described in section 3.1 followed by reengineering and analysing them according to the methodology shown in section 2.4. The reason for not patching directly on the device is the circumstance that when the patch is applied on the application, the patched class.dex will be stored as odex. Since the odex is specific to the code and device, it does not reveal the general approach of patching Since the code is modified directly, a static analysis is sufficient.

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

Before working on applications from the store, an application, including the LVL as Google's tutorial describes it [11], is created. The app is called *LicenseTest* in the following. This is done in order to test whether and how Lucky Patcher works on the most basic version. For Amazon's Kiwi DRM, the same application, with deactivated LVL, is uploaded to Amazon and injected with their DRM. In addition to LicenseTest,

other applications were analysed as well, in order to identify additional patterns and to analyse the result. These apps were chosen since they were already owned and tested to include the license verification library respectively Amazon DRM. The applications, which were approved to be included into the thesis by the developers, are Runtastic Pro[64], version 6.3, and Teamspeak 3[76], version 3.0.20.2, for the LVL and A Better Camera [5], version 3.35, for the Amazon DRM. The analysis for Samsung is done by using the example application since the library is implemented the same way into all applications used in. In contrast to Amazon this can assumed because it is done by the developer and not injected when uploading.
In addition to the analysis, the modified application is installed on different devices to evaluate the crack. The difference between the devices is the availability of the corresponding store as well as the presence of root and internet connection.
Lucky Patcher offers different modes to patch applications. At the end of patching in a modus the applied patterns can be seen like in figure 3.1 on the right. A pattern is a predefined sequence of byte code setting certain values. It is searched for and replaced by pre-fabricated code when cracking an application. These patterns are identified and evaluated after each mode is applied to the test set of applications.
These are the different modes and what Lucky Patcher descibes them as.

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

3.4 Patching Patterns

In order to identify the structure of the single patterns, the code of the original code was compared to the cracked output. The changes in the code were inspected on dex, smali and Java level with the tools explained in Section 2.4. After analysing same patterns in the different modes it was identified that these patterns can be summed up as one. The names of the patterns are taken from the patching result output in figure 3.1 on the right. The number next to the pattern indicates how often it was applied to the application. The patterns N*x* aim to circumvent the LVL while the Amazon and Samsung patterns are tailored to the respective store. This insight was gained in the analysis. In the context of the .dex, hexadecimal values like *0x0a* are simplified as *0a* for smoother reading. The opcodes can be taken from one of the many dalvik opcode documentations [61] and input arguments, as long as they are not constants, are called variable v*x* where *x* stands for the value. Since changing the dex code results in a different checksum of the code, the code has to be resigned. This changes appear in the diff of the dex files as well but are not explained since it does not have an impact on the functionality of the application itself.

Patch Pattern N1

Pattern N1 is present in all patching modes except the solo extreme mode. It tagets the *verify()* method of *LicenseValidator* class in the *com/google/android/vending/licensing/* folder which is responsible for dectypting and verifying the response from the license server [18].
When analysing the dex code, it is identified that the changes happen in the switch statement. The code block starts *00 02*, which stands for a sparse-switch, followed by *09 00*, which gives the number of elements. According to diff of the .dex file extracted, which can be seen in code snippet 3.1, element 1a and 0f are switched in their order, meaing 0f is now before 1a in the sequence of the switch.

```
@@ 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

Smali, which converts the .dex file to a human readable output, a code representation where result of opcode changes are easier assigned to methods. When converting the byte code to smali, a sparse switch struct with nine elements is created. Due to the internal mapping by the language, variables have different names. The swap of switch cases *0x1* und *0x2* can be seen in the diff of code snippet 3.2.

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

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

Code Snippet 3.2: Diff on Smali level for N1 pattern

The diff of Java, seen in code snippet 3.3, gives the best overview on the changed behavior of the application, since it does not contain jumps to abstract method names but includes the code as the developer intended it. Since the cracking pattern was also applied to the *LicenseTest* application, it is possible to compare the decompiled code to the original source code and it's original structure. The original library handles the *LICENSED* reponse the same way as *LICENSED_OLD_KEY*, since not having a breakpoint in the *LICENSED* switch case, the code subsequently continues in the *LICENSED_OLD_KEY* switch case and leaving it afterwards. The is only the reason if license keys are stored and allowed for verification, e.g. one time verification for future offline support. The entry point for a *NOT_LICENSED* response is follows after these two cases and is completed by a break to exit the switch case. When the pattern N1 is applied, the switch case statements of *LICENSED_OLD_KEY* and *NOT_LICENSED* are swapped, but not the code inside the block. This means that instead of *NOT_LICENSED* disallow the licensing process, it makes use of the *NOT_LICENSED* condition and allows access.

```
@@ 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)

As a results, pattern N1 voids the usecase of the switch in the method *verify()* since it always handles *NOT_LICENSED* response codes the same way as *LICENSED*.

Patch Pattern N2

As well as pattern N1, N2 is applied in all patching modes, except the solo extreme mode. It is more aggressive since it does not only attack the LVL library, but extends it attacks to other Google Mobile Service libraries, located at *com/google/android/gms/ads/*. The extended analysis of different applications showed the altering of custom libraries, which include code for Google inapp billing, like AnjLab's inapp billing library [22], located at *com/anjlab/android/iab/v3/Security*, in the FKUpdater, are modified as well.

This is collateral damage when applying the pattern to classes outside of the LVL libary caused by the possibility of moving the classes of the library to different packages. As long as the LVL package is untouched, pattern N2, similar to pattern N1, attacks the *LicenseValidator* class's *verify()* method.
The changes in the .dex file in code snippet 3.4 effect a simpler construct than pattern N1 does. Instead of of using the opcode *0a* for moving the result the result of the corresponding method to the variable 5 (source-destination order of *05*), it simply moves (opcode *12*) the value 1, which stands for true, to the variable 5 (*15*).

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

Code Snippet 3.4: Diff on Dex level for N2 pattern

Smali displays this in a more convenient manner as seen in code snippet 3.5.

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

Code Snippet 3.5: Diff on Smali level for n2 pattern

The impact on the Java code (see code snippet 3.6) is more complex. Now, instead of calling additional code in case the signature is verified, the additional code is always called. The reason for this structural change in code is the fact that in dex the if statement is split. In a first step, the signature is verified , while in the second step the result is moved to a variable and this variable is evaluated in the third step. Since the first step is not modified, it is still interpreted as a method by the decompiler. For the second and third step it is different. The reason is the decompiler since the variable is set with the constant value true and checked afterwards in an if evaluation. In order to simplify and remove superfluous code, the if statement is collapsed and the body is moved to outer scope.

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

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

The conclusion is that after applying this pattern the result of the verification of the signature is no longer included in the logic of *verify()*.

Patch Pattern N3

This pattern is different than the other patterns since there are two versions of it. Pattern N3 is present when using auto mode while pattern N3i is used in the inversed auto mode. The name N3i is chosen since it is used in the inversed auto mode. LuckyPatcher does not make a difference between them. They are combined under the same number since both attack the same logic inside the classes which define the policies. In case of *LicenseTest* application with the basic implementation of the LVL, these classes are the *APKExpansionPolicy* and *ServerManagedPolicy* in the *com/google/android/vending/licensing/* folder. Those two classes are examples of policies offered by Google [18] in which the *allowAccess()* is altered.
The goal of the two patterns is the opposite of each other. Both edit the value which is moved (opcode *12*) into variable 1, but while N3 is moving true (opcode *11*), N3i is moving false (opcode *02*) (see code snippet 3.7).

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

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

Code Snippet 3.7: Diff on Dex level for N3 pattern

As usual, the smali diff in code snippet 3.7 delivers a better presentation of the two patterns.

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

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

Code Snippet 3.8: Diff on Smali level for N3 pattern

The Java code in code snippet 3.9 is, again, more complex since the structure if the code in the .dex files is different than the one in Java. This is due to the fact that while in Java the *return* followed by a value is possible, in .dex files this statement is split in two parts. First, an internal variable is initialized with a standard value corresponding to the type of the return and in a second step the value is returned. The patterns modify the value initialization which results in the change of the return statement in the Java code.
Both patterns attack the class's *allowAccess()* method which evaluates whether the

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

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

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

verification result is according to the policy or not. Since this method is a place to change the logic in order to fool a cracking tool by changing the desired outcome from true to false, Lucky Patcher offers a way to deal with this implementation. This is the reason why the mode is called inverse auto mode since it fixes the return value to the opposite of the auto mode.

Patch Pattern N4

Pattern N4 was only applied once in the test sample and is part of the auto and auto inverse patching modes. The target of the pattern is the *LicenseChecker* class of the LVL which is responsible for initializing the license check in it's *checkAccess()* method [18]. When the pattern is applied, it replaces the method's equal zero check (*38*) with a check for not equal (*33*) as it can be seen in code snippet 3.10 and code snippet 3.11.

```
@@ 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

```
@@ 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

Since the code's syntax is changed when compiled from Java to dex, the decompiled code has a different sequence. In the source code of the LVL, the *checkAccess()* first checks whether the policy allows acces. When decompiled, it is checked for not equality first and the source code's equality block is moved to the else condition. This results in the initial equals zero condition and latter not equal test.

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

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

The result is that even though policy does not allow the access, this patterns modifies the *checkAccess()* method to act as it would have been allowed anyway.

Patch Pattern N5

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

As described before, the dex code is has a more complex decompiled Java code than the source code, especially in this case since instead of calling attributes of an object directly it stores them into variables before. In this decompiled code, the parsing of the result code is spoofed and not the value of the response data is stored but the response code is set to 0.

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

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

If this pattern would be applied alone, it would be able to circumvent the response code checks, but it is effective when combined with pattern N.

Patch Pattern N6

Pattern N6 is part of the extreme mode and attacks similar to the pattern N1, N2 and N5 the *verify()* method in the LVL's *'LivenseValidator* class.
The analysis of the .dex file in code snippet 3.14 reveals that the attack replaces two statements. The first modification is replacing the if equal zero (*38*) with setting a constant (*12*). This results in the argument, that should have been checked for equality with zero, is now the target variable of the constant move 0 (*0a*). Since the *if-eqz* opcode uses four touples, one of them is jump target which is set (*06*), it has to be removed to keep the syntax of the dex code valid. This is done by replacing the jump target with the *nop* opcode (*00*), which means no operation, and thus has no effect on the semantic of the code. The second change is the modification of the follow up if statement. Instead of checking whether the variable, which was originally checked for quality with zero, is equal (*32*) to another variable (*4a*), anoother variable is now compared to itself (*00*).

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

Code Snippet 3.14: Diff on Dex level for N6 patch

The change of the code structure is more visible in the code snipept 3.15. The *if-eqz* is replaced by the initialization of the p2, a nop operation and an empty line while the follow up statement has its two different arguments each exchanged with the same variable.

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

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

Code Snippet 3.15: Diff on Smali level for N6 patch

The analysis on Java level is more complex since removing the if statement destroys the original code herarchy and makes the syntax of the class invalid. Decompilers are built to work on correct syntax, in more complex code processes, the reverse engineered code cannot be created. In case of the *LicenseTest*, the decompiler was not successful as well, but the reversed code revealed that in the *verify()* method, right after the declaration of the variables, the second if statement of the two changed ones appeared This can be seen in the first part of the code snippet 3.16. The rest of the method cannot be decompiled and since translating from dex or smali to Java is hard to do by a human, this change cannot be analysed any further. This is the reason to look interpretation of code which was achieved in one of the apps, in this context it is called *IntelligentGadget* because the developer denied permission to name the application. In this application, the code was fully restored and the change was reflected in simple way. Instead of checking the response code whether it is not null (which cannot be seen in the code but is interpreted from the source code) and equal to the given value, it is checked whether it is not zero, which is not one of the specified values of the response code. For this reason the check for invalidity of the response code is never true and thus the termination conidition is never called. This simple reengineered code is the result of applying pattern N5 and pattern N6 in combination. The changes of these two patterns add up to a clean Java code as seen in the second part of the code snippet 3.16.

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

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

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

Patch Pattern N7

The final pattern for the LVL is pattern N7. It is applied in the extreme mode and as this mode indicates, it takes a harsh approach of applying itself. In addition to patching the LVL's *ILicenseResultListener onTransact()* method, which is the implementation fo the callback for interprocess communication which receives the async response from the license server [18], it applies it's patterns to all classes possible in the `com/android/`. It can be described as the bruteforce version of pattern N2 and thus should only applied in case the other modes are not successfully.
Similar to pattern N2, pattern N7 replaces the moving of the result (*0a*) of a function to the variable with initializing it with a constant (*12*). Since *move-result* moves it to

variable v0 (*00*) the resulting initialization sets v0 to zero.

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

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

The aim of this attack is to patch the *ILicenseResultListener* at any cost so *onTransact()* returns true independet of the result of *verifyLicense()* which is returned in the original implementation.

Overview for Patching the LVL

The summary of the LVL patterns and their use in the patching modes can be seen in table 3.1. The auto mode and inversed auto mode are applying patches at the important parts of the LVL. They are very efficient as long as the library is not altered. In contrast to the determined patching of the automatic modes, the extreme mode applies pattern covering a variaty of methods. This might cause instability, as seen in pattern N6, since it alters the syntax of the dex file.

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

Table 3.1: Patching patterns applied by each mode

Amazon Market Patch

Amazon does not have different pattern but only one patch which is called pattern in this context. Since the Kiwi library is injected by Amazon and cannot be customized by the developer as well as the structure of the library result in the fact that only on pattern is sufficient to circumvent the license verification mechanism The patch attacks two classes, one contained *com/amazon/android/licensing/b.java* and the other one contained in *com/amazon/android/o/d.java*. Since the injected library is obfuscated no meaningful names can be presented.

The Amazon pattern works similar to the LVL's pattern N4 and replaces the if equal zero opcode (*38*) with the opcode to check for not equality (*33*).

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

Code Snippet 3.18: Diff on Dex level for Amazon patch

In the original code, *if-eqz* just takes the second 0 of the input (*00*) for the comparison thus the first argument if the input is 0. As the result of patching, the first argument is taken as well and thus v0 is compared to v0.

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

Code Snippet 3.19: Diff on Smali level for Amazon patch

The Java code makes it easier to reflect of the changes byattack. Instead of checking in the *b.java* class, whether the response code is *LICENSED*, this statement of the if clause is always false and thus never called. The same canges are applied to the d.java class but since the variables are obfuscated it is hard to describe the exact behavior. After analysing the dependencies, it can be said that the function checks whether the given string is not null and then return true. After patching, the result is always true since the null check is always false as in *b.java*.

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

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

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

Samsung Market Patch

The Zirconia library of Samsung cannot be modified similar to Amazon and thus requires only one patch as well. The patch is applied on the *LicenseRetriever* and *Zirconia*

class in the *com/samsung/zirconia* package. Unlike Amazon's Kiwi DRM, Zirconia is not obfuscated and thus better to understand. The Samsung patch uses two patterns, called S1 and S2 in order to distinguish between them. While S1 is applied on both classes, S2 is applied twice but only on the *Zirconia* class.
While Pattern S1 replaces the arguments of the *if-eq d6* with *00* and instead of comparing two different variables, the variable is compared to itself, pattern S2 modifies the dex code to initialize a variable (*12*) instead of assigning it the result of a method *0a*.

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

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

Code Snippet 3.21: Diff on Dex level for Samsung patch

The result of this attack for pattern S1 is that the *if-eq* statement is always true while the result of methods affected by pattern S2 is always true.

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

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

Code Snippet 3.22: Diff on Smali level for Samsung patch

The abstract presentation of the result in Java code in code snippet 3.23 shows the resulting behavior. This means for *LicenseRetriever*'s *receiveResponse* method, that instead of checking the repsone code for validity, it is always accepted as valid. In the method *checkerThreadWorker* of the *Zirconia* class, the patch voids the check of the response code with the locally stored one. These behavior is the result of pattern S1. Pattern S2 works on the methods *checkLicenseFile* and *checkLicenseFilePhase2* of the *Zirconia* class. Instead to return the result of the license check, true always is returned.

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

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

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

The result of applying these patches is that not only the checks of the license files are voided and return always true but also reponse codes other than *LICENSED* accepted since they are neither compared to the accepted one nor to the stored one, whcih should be valid.

3.5 Conclusion and Learnings

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

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

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

lvl cracked app was always workign (test)

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

applies it directly to the librabry since first patching point could be the initial call, in case modified lvl patching initial call would be not enough since the on success block could contain important code (like ui creation) then it would be useless, target on specific points where decisions are made to alter as few code as possible since automated customizations have to be implemented to trick it make false checks to detect tampering -see- user patch

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

known bytecode patterns, replace with custom, makes mechanism useless

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

should not use one but many methods solution for current version of lucky patcher, future might be different, arms race scenario [58]

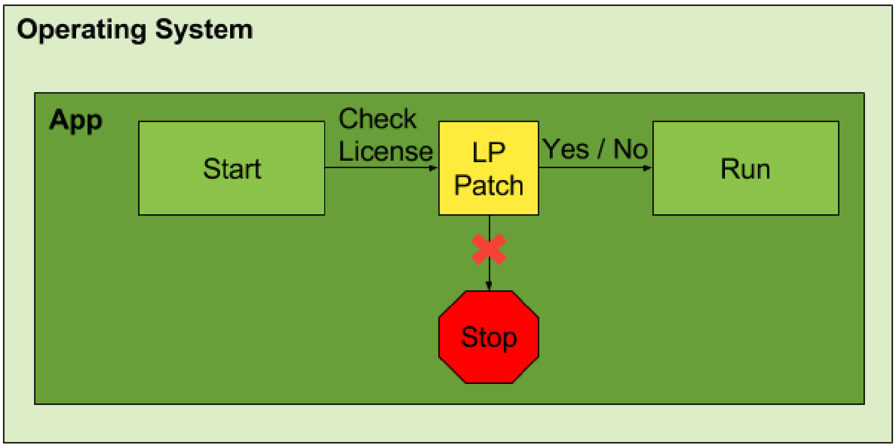

Figure 3.2: Abstraction of the current attack on the license verification mechanism

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

nur ja/nein test bzw ergebnis zuweisung und drauf folgender test kann IMMER geskippt werden, vergleich figure 2.12

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

4 Counter Measurements for Developers

Now that that the methodology of Lucky Patcher is analyzed, improvements to prevent the cracking app from circumventing the license check mechanism can be proposed to counter the identified attack. The first chapter will cover methods to improve the current implementation of the LVL, additional integrity tests to detect tampering and options to make the code more resistant against attacks to discover these additional tests. The second chapter proposes ways to make the the unary outcome when the license is verified more complex by introducing a content server and encryption. The third chapter takes a look at ART and Trusted Execution Environment (TEE) as environmental improvements to tackle this issue.

4.1 Extend Current Library

The first action against Lucky Patcher is to actively improve the spot its attacking which was identified and presented in figure 3.2. The goal is to prevent the automatical application of patching patterns as well as the execution in case tampering was detected. Since these checks can be voided after analysing the code manually and adding them to the patching procedure, obfuscation is introduced as a tool to make reverse engineering more complex.

4.1.1 Modifications on the Google LVL

Lucky Patcher automatic patching is reliant on applying patterns 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 [58]

this should be the first solution anyone does in order to actively fight luckypatcher automatischen angriff aushebenln indem andere struktur

a determined attacker willing to disassemble and reassemble code can eventully hack around the service developer can make hackers task immensely more difficult to the point where it is not worth the time lvl protects against casual piracy, users who try to copy apks directly from one device to another without purcahsing the applciation

modify to make difficult to apply common cracking techniques. make application tamper resistant oobfuscate to make it difficult tor everse engineer

[47]

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

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

CHARAKTERIZED IN DIFFERENT TYPES, tampering protection to detect attack, modifications to enforce additional work in order to crack, methods directly targeting reengineering and additional external features, can be stacked [58]

Modify the Library

The first step to actively fight against Lucky Patcher is to alter the license library. This can only be applied to the LVL since it is the only library where the source code is accessible for the developer. For Amazon and Samsung, this propose can be useful as well, but it is difficult to ship a high number of unique implementations as well as injecting it over the web interface cannot be done without additional work.
The main problem is that most developers include the LVL's source code unchanged and without modifications, lulled into a false sense of security by having some sort of license verification implemented into the application. Lucky Patcher profits from this misbelieve since patterns can be applied successfully without complciations.

google suggests additional improvements [47] modify the LVL in way that it is difficult for attack to modify the disassembled code and get a positive license check as result [47] provides protection against two different types of attack [47] - protects against attackers trying to crack app [47] - prevents attacks designed to target other applications (and stock lvl distribution itself) from being ported over to own app [47] goal is to increase the complexity of applcaition's bytecode and implementation unqiue [47]

three areas to focus on [47] core licensing library logic [47] entry/exit points of licensing library [47] invocation and handling of response [47]

core modifications, primary focus on two calsses which comprise the core of LVL - licensechecker and licensevalidator (see lucky patcher attacks them primary as well) [47] modify these two classes as much as possible, in any way possible, while retaining the original function of the application [47] some ideas, encouraged to be creative [47]

-replace switch with if, additional code between blocks => pattern nennen [47] - xor or ahsh functions to derive new values for any constants used and check for those instead CODE ISOMORPH ÄNDERN[47] -remove unused code, e.g. implement the policy cerification online with the rest of licensevalidator [47] -move entirety of LVL to own application's package [47] -spawn additional threads to handle different parts of license validation [47] -replace functions with inline code where possible => pattern nennen (moveresult) [47] - difficult to trace when decompiled, counter intuitive from traditional software engineering viewpoint, removing functions, hiding license check routines in unrelated code[47] - radical solution: kill app at first invalid response code, makes it impossible to alter follow up methods on reponse code, bad user experience

=> make lvl implementation unique, that is why examples are bad and own approach good [47]

for entry/exit [47] attack by attacker write counterfeit version of LVL thaht implements same interface, add additional arguments to licensechecker constructor and allow() and dontAllow() in LicenseCheckerCallback => pattern nennen [47]

attack calls in application to lvl, e.g. dialog and attacker comments out, add different activity that handles informing of invalid license and temrinates original activity, add finish() to other parts of code which is executed in case original one gets disabled, or set a timer that will cause termination after timeout [47] postpone licensecheck sine attackers expect instant license check during launch [47]

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

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

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

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 this are all improvements which do net ensure no 100% safety, at least can protect against autopatching, additional work for attackers but as it seams developers do not like to modify as seen in analysis since all apps are somehow patchable

Native Implementation

Lucky Patcher's automatic patching modes, as described in chapter 3m target the application's .dex file. In order to thus removing it from this file makes it more complicated to crack the application sucessfully. Since Android supports the Native Development Kit (NDK), it is possible to implement parts of the application using native code. Usually the NDK targets CPU intensive taks, such as game engines and signal processing, but it can be used for any other purposes as well. Google suggests using it only if necessary since it increases the complexity of an application, which, in case of the license verification library, is desired [12].

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

this way luckypatcher has to create a custom recipe for your application since custom recipes can inject .so files at first a good protection against luckypatcher since the automatic patching discussed in does not work for the app any longer, but not protection of luckypatcher because it supports custom patching when analyzing luckypatcher there can be .so files be found which are used to replace existing ones in order to patch fucntionality

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

elemente für app in native initiieren, damit man nicht einfach deaktivieren kann nur gut wenn wichtige funktionen der app enthält native code is difficult to be decompiled, can be checked for tampering when loading it

4.1.2 Tampering Protection

mechanisms should work for amazon/lvl/samsung

Environment and Integrity Checks, wenn die umgebung falsch ist, kann die app verändert werden. deswegen von vornherein ausschließen, dass die bedingungen dafür gegeben sind.[58]

in order to remove/disable lvl they have to modify the code unless done precisely can be detected by code (signature muss geändert werden von lucky um dex fiel gültig zu halten)[47]

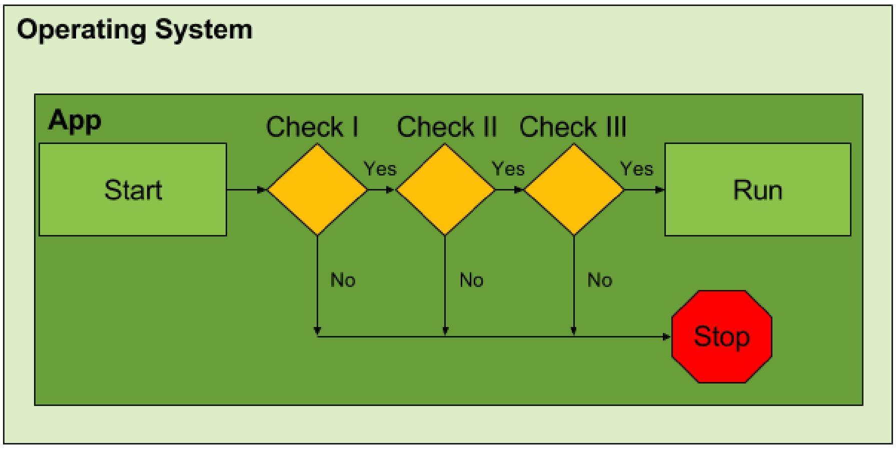

Figure 4.1: Introduction of additional tests to check environment and integrity of the application

his does not stop luckypatcher in anyway but it stops the app from working in case the environment is not suitable 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.

all tampering counter measurements have kind of the same pattern, boolean check, simple method == simple fix, can be nulled easily when code is known, just as easy to crack as LVL when you know the code, but attackr has to invest some time to understand code and to build counter measurement, in addition with Section 4.1.3 this can get annoying, evtl create native versions because harder to crack even though it is simple it adds a little bit extra work to attack and when cobined this grows exponential automatischen angriff aushebenln indem andere unbekannte tests die erst entdeckt werden müssen add additional test, have to be cracked as well, first attacker has to realize them and find them all tampering counter measurements have kind of the same pattern, boolean check, simple method == simple fix, can be nulled easily when code is known, just as easy to crack as LVL when you know the code, but attackr has to invest some time to understand code and to build counter measurement, in addition with Section 4.1.3 this can get annoying, evtl create native versions because harder to crack even though it is simple it adds a little bit extra work to attack and when cobined this grows exponential

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.

Debuggability

Enabling debugging allows the developer to use aditional features for analysing the app at runtime, like printing out logs [16]. This features are used to gain information about the flow of the app and to reenginer functionality. From the results of this analysis weak points in the application can be identified and custom patches for cracking tools can be derived from them. This is the reason for disabling the debug flag in release builds on the application stores, but since the flag for debugging is included inside the .dex file, it can be enabled by attacks In order to prevent attackers taking advantage from this possiblity, the developer should check whether this flag is activated und thus the application is tampered.
Code snippet 4.1 is an example for an implementation of this check. The debug flag

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

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

    return debuggable;
}
```

Code Snippet 4.1: Example code for checking for debuggability can be aquired from the application information as seen in line 15. In case the debug is set, and thus the application is tampered, the process is killed in line 18.

Root

Root is the foundation to cracking applications and thus when the device is rooted it possible to alter the application. The developer can check whether root is available on the device and evtually exclude users accordingly. When using this feature, the developer has to communicate the users the reasons for this strict policy since there are a lot of users who use root for other reasons than cracking applications. Google has introduced a similar APK, called SafetyNet [51], which is said to be used in security critical applications like Android Pay [14] [63].
Since root is achieved by moving the *su* file to the file system, the application can search

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

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

        return result;
    }
```

Code Snippet 4.2: Example code for checking for root

in the common locations for it. In case the search is successful, the execution of the application can be terminated.

Lucky Patcher

Having Lucky Patcher installed is an indicator that the user is attack license verifications libraries. The check is not as strict as the one for root and exlucding the user from executing is builds upon a foundation. It can be extended to detect additional unwanted applications by adding their package name to the check [82]. Some custom Android versions already include a library called *AntiPiracySupport* [36] which is used to remove and blacklist piracy applications.
As shown in code snippet 4.3, the check tries to aquire whether the Lucky Patcher

```
    public static boolean checkInstall(final Context context) {
        boolean result = false;
        String luckypatcher =
                // Lucky patcher 6.0.4
                "com.android.vending.billing.InAppBillingService.LUCK",
        };

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

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

        } catch (final PackageManager.NameNotFoundException ignored) {
        }

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

        return result;
    }
}
```

Code Snippet 4.3: Partial Listing

package is installed. In case information is available and thus the application is installed, the check stops the application.

Sideload

APKs can be cracked on a different device, transfered and installed on the device, so checking for root and Lucky Patcher are not enough. Usually, applications, which

include a license verification library, are purchased from the corresponding store. Installing them from other sources is a sign for piracy. For this reason, developers should enforce installation from trusted sources to ensure that the application is purchased as well.
The code snippet 4.4 shows the implementation for the stores in scope for the thesis.

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

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

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

        return result;
    }
```

Code Snippet 4.4: Partial Listing

Additional stores can and should be added in case the developer decides to offer the application in another store.
This feature should be implemented with caution since Google notes that this method relies on the *getInstallerPackageName* which is neither documented nor supported and only works by accident [47].

Signature

`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 bytecode 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. [58] !!!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?

da signatur geändert ist, funktionieren dinge wie google maps nicht mehr -> build in tampering protection

Flow Control

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

4.1.3 Obfuscation

man versucht sich vor neuen angriffen zu schützen indem man reengineering zeitintensiver macht
wenn anderes zb nicht geht weil algorithmus in der app, ohne redesign, wenn man auf kapiutel 1 sitzen bleibt, kein kontent server möglich, smartcard nicht einbauen will weil extra hardware/kosten,...

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

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

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

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

    return result;
}
```

Code Snippet 4.5: Partial Listing

ref kapitel 1 macht gg lucky sicher, aber crakcbar wenn manuell, was kann man dagegen tun

now that the lvl is modified and the environment is enforced, the next goal is to prevent pirates from even starting to analyze the applcation, automatischer crack nicht mehr funktioniert, muster, andere mechanismen erkennt er nicht

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

[50] it is not possible to 100 percent evade reengineering, but adding different methods to hide from plain sight of reengineering tools reengineering cannot be vermiede best is to apply technqiues to make it as hard a possible [58]

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

This is my real text! Rest might be copied or not be checked! SOLUTION 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
[50]

a few dex obfuscators exist, with different approaches proguard or sdex, rename methods, field and calss names – break down string operations so as to chop hard coded strings or encrypt – can use dynamic class loading (dexloader classes to impede static analysis) can add dead code and dummy lopps (minor impact of performance) can also use goto into other onstructions

[54]

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

[58]

first line of defense, can be applied without much extra workload will not protect against autoamted attack, does not alter flow of program makes more difficult for attackers to write initial attack removing symbols that would quickly reveal original structure number of commercial and open-source obfuscators available for Java that will work with Android

be aware that certain methods cannot be obfuscated, even with advanced tools e.g.

on create cannot be renamed since it needs to remain callable by the android system avoid putting license check code in these methods since attackers will be looking for the lvl there [47]

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)

angriff beschreiben - library öffentlich - reengineering (ref meien tools)

davor schützen, jeden beliebigen test durch unäre antwort ersetzen, um das zu umgehen höher level als die license verifcation

=> tests die man auf unär modifizieren kann sind schwerer zu lokalisieren, hilft gg reengineering

in theory a good addition to the security of the application, but against luckypatcher directly since it works on java level in order to disguise the way the code works it enforces increased initial effort an attacker has to spend in order to understand the code and thus reduces the likelihood of attackers to being motivated to crack the application in practice obfuscators are limited due to: reliance on android framework apis (remain unobfuscated) applications can be debugged jdwp and application debuggability at the java lvl can reveal information about the software popular enough obfuscators (dexguard) have deobfuscators professional tools cost a lot of money, you have to be commercial or have a real good idea in order to be worthwhile

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 [50]

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

[58]

identifier mangling, ProGuard uses a similar approach. It uses minimal lexical-sorted strings like a, b, c, ..., aa, ab, original identifiers give information about interesting parts of a program, Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed -see- neutralizing these information in order to prevent this reduction, remove any meta information about the behavior, meaningless string representation holdin respect to consistence means identifiers for the same object must be replaced by the same string, advantage of minimizing the memory usage, e development process in step "a" or step "b"
string obfuscationa, string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog, information is context, other is information itself, e.g. key, url, injective function and deobfuscation stub which constructs original at runtime so no behaviour is changed, does not make understanding harder since only stub is added but reduces usable meta information
[67]

ProGuard is an open source tool which is also integrated in the Android SDK, free ProGuard is basically a Java obfuscator but can also be used for Android applications because they are usually written in Java // feature set includes identifier obfuscation for packages, classes, methods, and fields was kann er noch? -see- Besides these protection mechanisms it can also identify and highlight dead code and removed in a second, manual step Unused classes removed automatically by ProGuard. easy integration[4]

optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code merges identical code blocks performs optimiztations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [41] [78]

Dexguard

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

commercial Android obfuscator for bytecode and source code various techniques including strings encryprion, encrypting app resources, tamper detection, removing logging code [50]

son of proguard, does everything that proguard does its a optimizer and shrinekr, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), applciation tamper protection, removes debug information may increase dex size, memory size; decrease speed [78]

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

protects apps from reverse engineering and ahckign attacks makes apps smaller and faster specialized fr android protects code: obfuscation, hides sensitive strings,keys and entire algorithms [45]

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 [78]

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

[2] [3]

4.2 Complex Verification

A part of the license verification's insecurity is the fact that not only the possible outcome is binary, either the license is verified or it is not, but it is also possible to predict the outcome since in the end, only one result is accepted. This is mechanism can be easily attacked by cracking applications since they only have to modify in a way that only the result is unary in the end and thus, despite the input, always accepted. In order to fight this major flaw, more complex mechanisms to replace the old license verification are proposed. The first proposal is the introduction of content server which allows to move part of the functionality, like the license verification, from the application to a server and thus not accessible by attackers. The second proposal is to introduce encryption and replace the binary mechanism by an unpredictable one.

4.2.1 Content Server

Moving the license verification to a server is one approach to fight the shortcomings of the license verification libraries. By introducing service managed accounts, the user has to login on a server in order retrieve content for the application. Since the applications logic is on a server, Lucky Patcher is not able to manipulate the verification logic. The implementation can be described with the application Spotify [69] as a

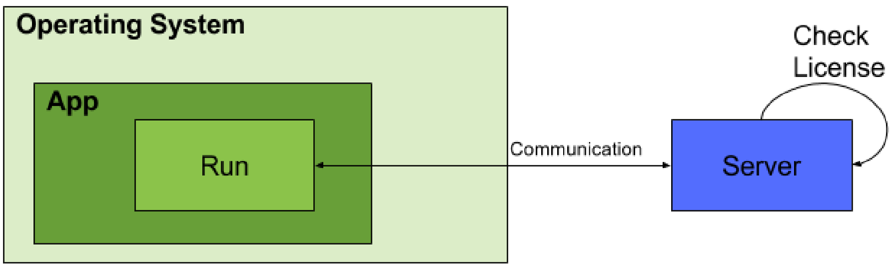

Figure 4.2: Abstraction of an application and a content server

reference. Instead of using local license verification, the user has to enter his credentials, which are send to the server. In case the credentials are valid, the user is logged into the application. The content, music in this case, is no longer on the phone itself, but streamed from the server. In case a attackers circumvent the user verification, they are facing a second layer of security since it is not possible to access the content on the server without a verified account. Thus attacks on the applications are not simple anymore.
This would be a perfect solution if there were not downsides as well. The first problem is that this model cannot be applied universally. This means that it must be possible to extract parts of the application's logic and implement them on a server. The second one are the additional resources needed. When realizing parts of the application on a server, not only knowledge and money is needed for the server, an additional application for the server has to be created as well. Not every developer can handle this additional workload. The third problem is the resulting always online necessaty which limits the

freedom of users and might make the application less accepted.
Nevertheless, in case this implementation can be realized, it is an almost safe solution. As a byproduct, the protection of the IP can be achieved as well by moving the core algorithm, when possible, to server. This prevents attackers not from only using the application for free, but also from reconstructing the the core functionality and implementing it somewhere else and thus the application offers less value to attackers.

4.2.2 Encryption

Another approach is the introduction of more complex license verification mechanism by using encryption. The advantage of encryption is the unpredictable outcome of an input which is nto spoofable anymore. Before taking advantage of encryption, it has to be decided what content should be encrypted and where the key should be stored.

Encryption

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

Resource Decryption

The first approach is to apply encryption on the application's static resources. This includes the application's hard coded strings as well as image assets. Whenever a resource is needed, it has to be decrypted first. The increase in security comes at the cost of decreased performance. As long as application critical strings, like server addresses are encrypted, the application is unable to work. This is not the case when no such system relevant strings are included. THen the application will work as usual, but the user experience will be not sufficient because all strings are still encrypted and have no meaning. Figure 4.3 shows the abstract implementation of resource decryption.

Action Obfuscator

The second approach is using encryption as a kind of obfuscation. The idea is to have a method, which is called by each method, to deligate all calls according to an encrypted parameter. The attacker can try to guess which method call is linked to each method, but when additional encryption for the parameters is applied and decryption is used in the target method, it is very hard to crack the logic without modifying a lot of code. The encryption can combined with additional manipulation of the objects in order to make the simple removing of encryption not enough. This methodology is most

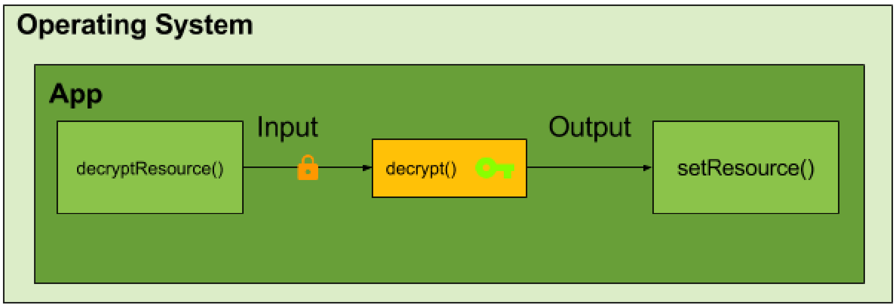

Figure 4.3: Encrypted resources which have to be decrypted on startup

effective on non boolean objects. An abstract presentation of the mechanism can be seen in figure 4.4.

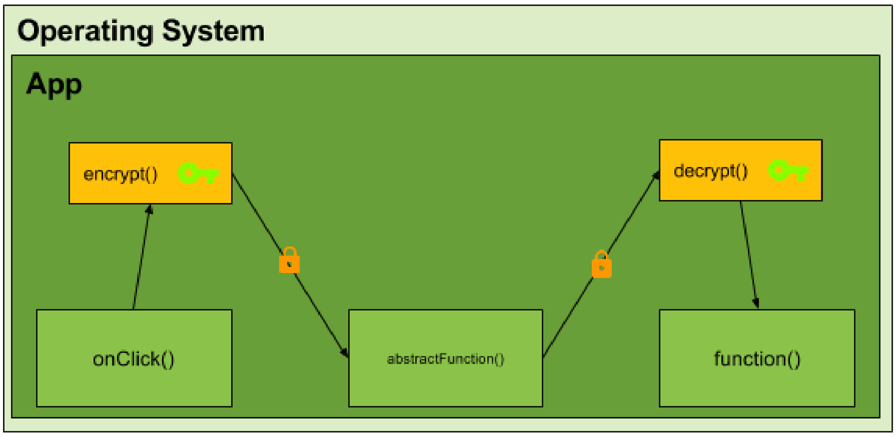

Figure 4.4: Encrypted actions to obfuscate dependencies

Communication Decryption

The third approach

additional secure feature applied to content server

combined with device specific key, accounts cannot be shared anymore (either hacked or shared libratly)

`https://source.android.com/devices/drm.html` store key outside the app and decrypt content, works only for apps which can work with content server, focuses on the security of content on the phone

delivering content is allowed but the deliverer wants to be sure that nobody steals it from the phone

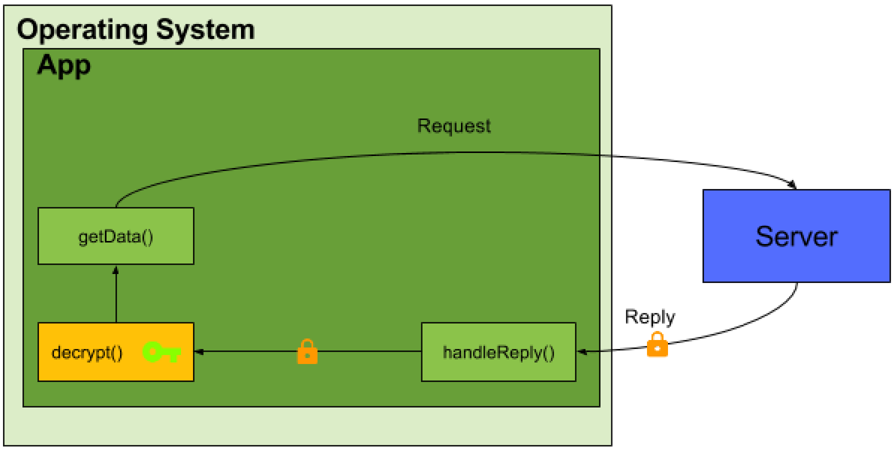

Figure 4.5: Encrypted communication with a server

Encryption

The encryption approaches presented are only as strong as the protection of key for decryption.

two important quesions have to be asks what will be target of encrypion where will the eky be retrieved and where is it stored, difficult since online offline

Secure Element

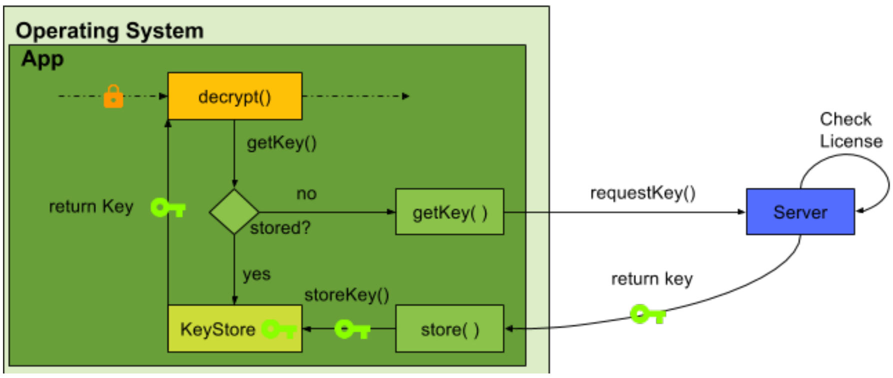

Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

sicher gegen lucky patcher, da runtergeladen werden muss und gespeichert werden muss, und man muss erstmal manuel extracten und dann verteilen

Secure Element

idea: store key at secure place where it is not exposed but delivers decrypted output for an input, best suitable for this task smartcard

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 [70] with power as low as 25MHz complexe calculations would take too much time

simple tasks (beschreiben)

unbestechlich, jedoch boolean abfrage immer crackbar, deswegen andere sachen machen zB verschlüsseln

encrypted strings from the application can be decrypted by the secure element store property settings on it

encryption key can be stored on

no complexe tasks since low power

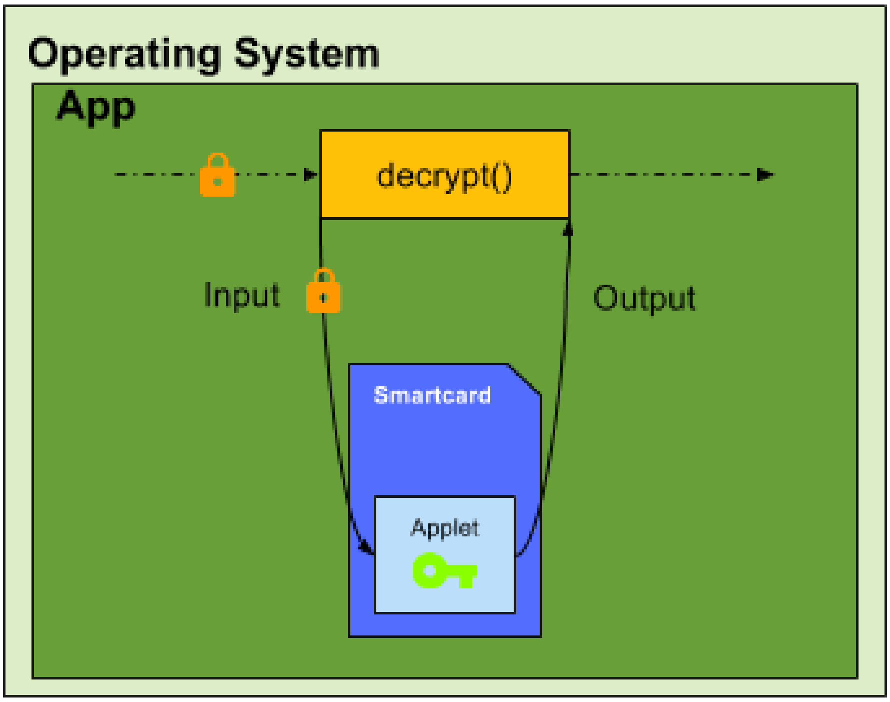

Figure 4.7: Decryption by using a smartcard

same as internet service but extra hardare has to be bought people are lazy and do not want to have extra hardware when integrated into phone, it will take a long time until each phone supports it, e.g. was not able to use it on Nexus 6P and Nexus 7, Linux did not recognize it but mass storage was enabled, Nexus7 said OTG available but it did not work many different implementation fragmentate the market, there is not one single solution to focus on and push for market wide accepted solution solution which are out there have major security flaws, smartcard itself can be attacked

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

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

SD Association http://www.androidpolice.com/2016/02/22/with-smartsd-the-sd-association-wa

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

4.3 Extend Environment

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

level tiefer

4.3.1 Trusted Execution Environment

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

WAS IST ES? WAS MACHT ES? WIE IMPLEMENTIERT MAN?

promoted as be all end all solution for mobile security in theory isolated processing core with isolated memory, cannot be influenced by the outside and runs with priviliged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitve processing offloaded to protect information from malware perfect wish: secure chip to process software that malware should not access, security related stuff like bankin, encryption

example Trustzone, Knox

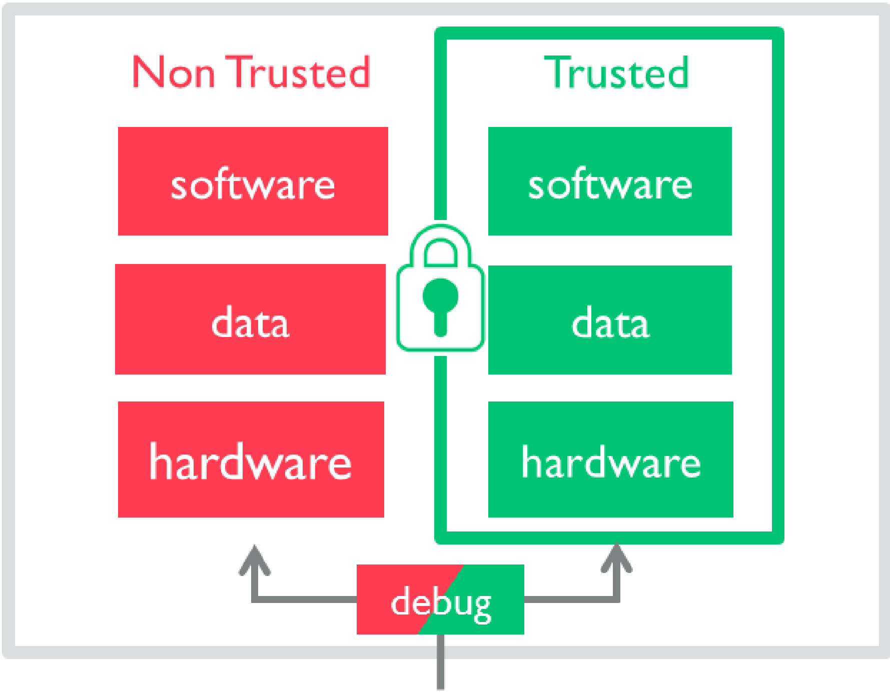

Figure 4.8: tee [26]

[31][26]

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

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

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

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

protection validation can be done by either using qualcomms or writing a custom

one giant box, error by one has impact on all others

[31][26]

not accessible to anybody in general many different solutions, focus should be on one unique standard in order to fix the problems and make compatibility , google already started by integrating features of samsung's knox into android lollipop [65]

4.3.2 ART

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

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

already on the way, cannot be done from one day on the other, but right now not a protection against luckypatcher, will only be a solution when art code included in apks but why not now? Evaluation Why is Android not all ART now? Your applications still compile into Dalvik (DEX) code, Final compilation to ART occurs on the device, during install, Even ART binaries have Dalvik embedded in them, Some methods may be left as DEX, to be interpreted, Dalvik is much easier to debug than ART
[54]

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

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

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

When creating odex on art it is directly put into art file
[54]

5 Conclusion

research and also a valuable market for companies
Because source code can be easier recovered from an application in comparison to x86, there is a strong need for code protection and adoption of existing reverse engineering methods. Main parts of Android application functionalities are realized in Dalvik bytecode. So Dalvik bytecode is of main interest for this topic
Also, the Android system does not prevent modification of this bytecode during runtime, This ability of modifying the code can be used to construct powerful code protection schemata and so make it hard to analyze a given application.
[67]

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

lvl popular but broken, has not done much since beginning of known issues [58]

auch wichtig weil wenn crackable dann upload zu stores und dann malware http://www.hotforsecurity.com/blog/mobile-app-development-company-fights-off-android-malware-with-obfuscation-tool-3717.html

5.1 Summary

jedes chapter beschreiben

5.2 Discussion

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

shortly after start insufficient reilience against reverse engineering, not explusivly to lvl thus shift from lvl protection to general protection against reverse engineering, decompilation and patching eternal arms race no winning solution against all cases, jsut small pieces quantitative improvement no qualitatively improve resilience limited to quantitative resilience, matter of time until small steps generate more work for reengineering, ggf lower motivation for cracker only matter of time until patching tools catch up, completely new protection schemes need to be devised to counter those [58]

not a question of if but of when bytecode tool to generate the licens elibrary on the fly, using random permutations and injecting it everywhere into the bytecode with an open platform we have to accept a crack will happen [49]

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

alles hilft gegen lucky patcher auf den ersten blick, jedoch custom patches, welche Lucky Patcher anbietet[58], können es einfach umgehen, deswegen hilft nur reengineering schwerer zu machen viele piraten sind nicht mehr motiviert wenn es zu schwer ist every new layer of obfuscation/modifcation adds another level complexity

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

5.3 Future Work

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

lvl has room for improvement art promising but not root issue, dex is distributed and art compilation to native on device needs to become relevant so developers can release art only apps, native code and no issue with reverse engineering stop/less important

until lvl see major update custom improvements have to be done [58]

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

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

google hat schon sowas wie google vault

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

List of Figures

List of Tables

List of Code Snippets

Bibliography

[1] Alastair Beresford, et al. *All Vulnerabilities*. URL: http://androidvulnerabilities.org/all (visited on 01/24/2016).

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

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

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

[5] Almalence Inc. *A Better Camera*. URL: http://www.amazon.de/Almalence-Inc-A-Better-Camera/dp/B00HUP8UZA (visited on 02/17/2016).

[6] Amazon. *Amazon Developer Service*. URL: https://developer.amazon.com/ (visited on 02/02/2016).

[7] Amazon. *Amazon Send Developers a Welcome Package*. URL: http://www.androidheadlines.com/2010/10/amazon-send-developers-a-welcome-package.html (visited on 01/19/2016).

[8] Amazon. *Introducing Amazon Appstore for Android*. URL: http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1541548 (visited on 01/19/2016).

[9] Android. *ART and Dalvik*. URL: https://source.android.com/devices/tech/dalvik/index.html (visited on 02/15/2016).

[10] Android. *Configuring ART*. URL: https://source.android.com/devices/tech/dalvik/configure.html (visited on 02/15/2016).

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

[12] Android Developers. *Android NDK*. URL: http://developer.android.com/tools/sdk/ndk/index.html (visited on 02/21/2016).

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

[14] Android Developers. *Checking Device Compatibility with SafetyNet*. URL: https://developer.android.com/training/safetynet/index.html (visited on 02/21/2016).

[15] Android Developers. *Dalvik Executable format*. URL: https://source.android.com/devices/tech/dalvik/dex-format.html (visited on 02/02/2016).

[16] Android Developers. *Debugging*. URL: http://developer.android.com/tools/debugging/index.html (visited on 02/21/2016).

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

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

[19] Android Developers. *Setting Up for Licensing*. URL: https://developer.android.com/google/play/licensing/setting-up.html (visited on 01/18/2016).

[20] Android Developers. *Settings.Secure*. URL: http://developer.android.com/reference/android/provider/Settings.Secure.html (visited on 02/16/2016).

[21] Android Developers Blog. *Announcing the Android 1.0 SDK, release 1*. URL: http://android-developers.blogspot.de/2008/09/announcing-android-10-sdk-release-1.html (visited on 02/15/2016).

[22] AnjLab. *A lightweight implementation of Android In-app Billing Version 3*. URL: https://github.com/anjlab/android-inapp-billing-v3 (visited on 02/18/2016).

[23] I. R. Anwar Ghuloum Brian Carlstrom. *The ART runtime*. URL: https://www.youtube.com/watch?v=EBlTzQsUoOw (visited on 02/02/2016).

[24] APKSFree.com. *diff*. URL: http://www.androidapksfree.com/app/blackmart-alpha-latest-version/ (visited on 02/11/2016).

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

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

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

[28] D. Bornstein. *Dalvik VM Internals*. URL: https://sites.google.com/site/io/dalvik-vm-internals (visited on 02/02/2016).

[29] 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).

[30] 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).

[31] 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).

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

[33] W. Choi. *A mini project to customize existing AXML library*. URL: https://github.com/wtchoi/axml (visited on 02/17/2016).

[34] 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).

[35] 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).

[36] ContentGuard. *AntiPiracySupport*. URL: https://github.com/ContentGuard/AntiPiracySupport (visited on 02/21/2016).

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

[38] Dex Protector. *Dex Protector*. URL: https://dexprotector.com/ (visited on 02/15/2016).

[39] 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).

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

[41] 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).

[42] GitHub. *Atom - A hackable text editor for the 21st Century*. URL: https://atom.io/ (visited on 02/17/2016).

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

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

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

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

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

[48] 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).

[49] 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).

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

[51] J. Kozyrakis. *AntiPiracySupport*. URL: https://koz.io/inside-safetynet/ (visited on 02/21/2016).

[52] M. Kroker. *App-Markt in Deutschland 2014: Umsätze im Google Play Store erstmals größer als bei Apple*. URL: http://blog.wiwo.de/look-at-it/2015/02/25/app-markt-in-deutschland-2014-umsatze-im-google-play-store-erstmals-groser-als-bei-apple/ (visited on 01/16/2016).

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

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

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

[56] S. R. Lingala. *Zip4j - Java library to handle ZIP files*. URL: http://www.lingala.net/zip4j/ (visited on 02/17/2016).

[57] 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).

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

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

[60] Oracle. *JAR File Specification*. URL: http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html (visited on 02/15/2016).

[61] G. Paller. *Dalvik opcodes*. URL: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html (visited on 02/18/2016).

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

[63] Pulser_G2. *A Look at Marshmallow Root and Verity Complications*. URL: http://www.xda-developers.com/a-look-at-marshmallow-root-verity-complications/ (visited on 02/22/2016).

[64] 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).

[65] Samsung. *A closer look at KNOX contributing in Android*. URL: https://www.samsungknox.com/en/androidworkwithknox (visited on 01/24/2016).

[66] Samsung. *How to protect your app from illegal copy using Samsung Application License Management (Zirconia)*. URL: http://developer.samsung.com/technical-doc/view.do?v=T000000062L (visited on 01/19/2016).

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

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

[69] Spotify Ltd. *Spotify Music*. URL: https://play.google.com/store/apps/details?id=com.spotify.music&hl=de (visited on 02/21/2016).

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

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

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

[73] 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).

[74] 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).

[75] 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).

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

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

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

[79] ubuntuusers. *diff*. URL: https://wiki.ubuntuusers.de/diff (visited on 02/02/2016).

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

[81] WugFresh. *Nexus Root Toolkit v2.1.4*. URL: http://www.wugfresh.com/nrt/ (visited on 02/16/2016).

[82] Za hiD. *ANTI-PIRACY SOFTWARE ACTIVATED SOLVED*. URL: http://android-onex.blogspot.de/2015/07/anti-piracy-software-activated-solved.html (visited on 02/21/2016).