DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Johannes Neutze, B. Sc.

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Analyse von Android Crackingtools und Untersuchung geeigneter Gegenmaßnahmen für Entwickler

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

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

Munich, March 15, 2015 Johannes Neutze, B. Sc.

Acknowledgments

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

Assumption

- it master
- knowledge of programming, java, android
- android apps, distribution

Abstract

http://users.ece.cmu.edu/~koopman/essays/abstract.html Motivation Problem statement Results Approach Conclusions

Android is the biggest mobile Operating System (OS). This makes it target of software piracy. Developers cannot protect their Intellectual Property (IP) because implemented license verification mechanism are attacked and easily voided by cracking tools. This thesis analyses the cracking tool Lucky Patcher and presents the way it works. The findings are that the attacks are executed by modifying different parts of the code. Since the response from the license server is always binary, Lucky Patcher does not have to change the library but attacks the decision points. The result of the evaluation is always ignored and the code is executed as if valid. The approach to counter Lucky Patcher is either an unique implementation of the library, a no longer binary decision or improved environment. As long as the code can be analysed it can be altered.

- priacy problem
- android different approaches
- luckypatcher attacks
- does not change methods, but whether their outcome is included
- not a simple solution, uniqueness or outsourcing best protection
- a lot to do on android

Contents

Glossary

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

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

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

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

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

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

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

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

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

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

Acronyms

.dex Dalvik EXecutable file.

.jar Java Archive.

.odex Optimized Dalvik EXecutable file.

ADB Android Debug Bridge.

ADT Android Developer Tools.

AOT Ahead-Of-Time.

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.

OTG USB On-The-Go.

SDK Software Development Kit.

SE Secure Element.

TEE Trusted Execution Environment.

VM Virtual Machine.

1 Introduction

1.1 Licensing

Software Licensing is the legally binding agreement between two parties regarding the purchase, installation and use of software according to its terms of use. It defines the rights of the licensor and the licensee. The goal is to protect the software creator's IP or other features and enable him to commercialize it as a product. It defines the boundaries of usage for the user and prevents him from illicit usage [80].
Software licenses come in different variants. They range from open source over usage for a limited time to usage of a limited set of features. Since using the full feature set of software might be bound to paying a royalty fee, these software is often subject of piracy. In order to prevent unauthorized use, mechanisms to enforce the legal agreementare implemented. This includes Digital Right Management solutions which deny access to the software in case of a wrong serial key or unregistered account.

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

1.2 Motivation

Licensing is also present in Android. With a market share of almost 82.8% in Q2 of 2015 [52] it is the most used mobile OS. According to Google, this market share translates to over 1.4 billion active devices in the last 30 days in September 2015 [32]. This giant number of Android devices is powered by Google Play [49]. Google's marketplace offers different kinds of digital goods, as applications, music or movies, but also hardware. In the application section of Google Play user can chose from over 1.6 million applications for Android [85]. In 2014 Google's marketplace overtook Apple's Appstore, which had a revenue of over 10 billion back in 2013, and became the biggest application store on a mobile platform [58].
The growth comes with advantages. Some time ago developers only considered iOS as a profitable platform and thus most applications were developed for Apple's OS

only or at least first. Now, with Android's overwhelming market share, they focus heavily on Android [68]. But this also creates a downside. The expanding market for Android, offering many high quality applications, draws the attention of software pirates. Crackers do not only bypass application's license mechanisms and offer them for free. Redirecting cash flows or distributing malware using plagiates is an lucrative business model as well.
Android developers are aware of the situation [87] and express their need to protect their IP on platforms like xda-developers [74] or stackoverflow [78]. Many of the developers have problems with the license verification mechanism and name *Lucky Patcher* as one of their biggest problems [79].

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

1.3 Related Work

Lucky Patcher and license verification have already been topic of scientific work.
Patrick Bernhard takes a look at license verification an in-app billing attacks in his master's thesis *A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks*. He comes to the conclusion that the libraries need an overhaul since they are easy to circumvent and have not been updated for a long time. This shows the urgency on further investigation on this topic.
Marius-Nicolae Muntean's master's thesis *Improving License Verification in Android* presents an analysis of techniques to crack Android's license verification and implements a new approach. He introduces multiple general strategies, such as obfuscation and dynamic code generation, to fortify code. In the end he uses the insights from the analysis to suggest countermeasures and their effects. A similar approach is chosen for this thesis.
In addition to

2 Foundation

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

2.1 Software Piracy

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

2.1.1 Developers

Piracy is a big problem for developers as seen in figure 2.1. The developer loses direct revenue when his IP is stolen and redistributed by a pirate without the developer's involvement. In case the application is offered for free, users do not have to pay in order to download it and no revenue is created. It is even worse when the pirated application has to be purchased in another app store. In this case the pirate will get the profit which should be the developer's.
But revenue is not only lost when the application can be downloaded for free. The pirate is also able to dry up follow up revenue by modifying the application itself. There are two main types of indirect revenue. The first type are in-app purchases. They are a popular source of income for so called freemium applications or lite versions of applications. In case of the the freemium app, the download is for free and includes all features. The developer makes the money of in-app purchases like cosmetic modifications or in game currency. The lite version application is a little bit different. The download is free as well but the application comes with a restricted feature set or limited time of use. In order to take advantage of the full feature set the user can buy the pro license via an in-app purchase. Apps can include a mix or various degrees of theses types. Pirates can disable the transaction of the payments for the

in-app purchase. This makes the in-app purchases cost no money and thus no earnings are generated for the developer.
The second type of indirect revenue is generated by showing in-app advertisements. When this feature is implemented, advertisements are shown inside the application and the developer is paid by views and clicks on the advertisements. Earnings generated by an applications are assigned according to the included Ad Unit ID [48]. When an application is pirated, this ID can be replaced by the pirate's ID. Future revenues generated by advertisements will not be assigned to the developer but to the pirate.
Beside monetary issues, additional problems arise when the application is moved to a

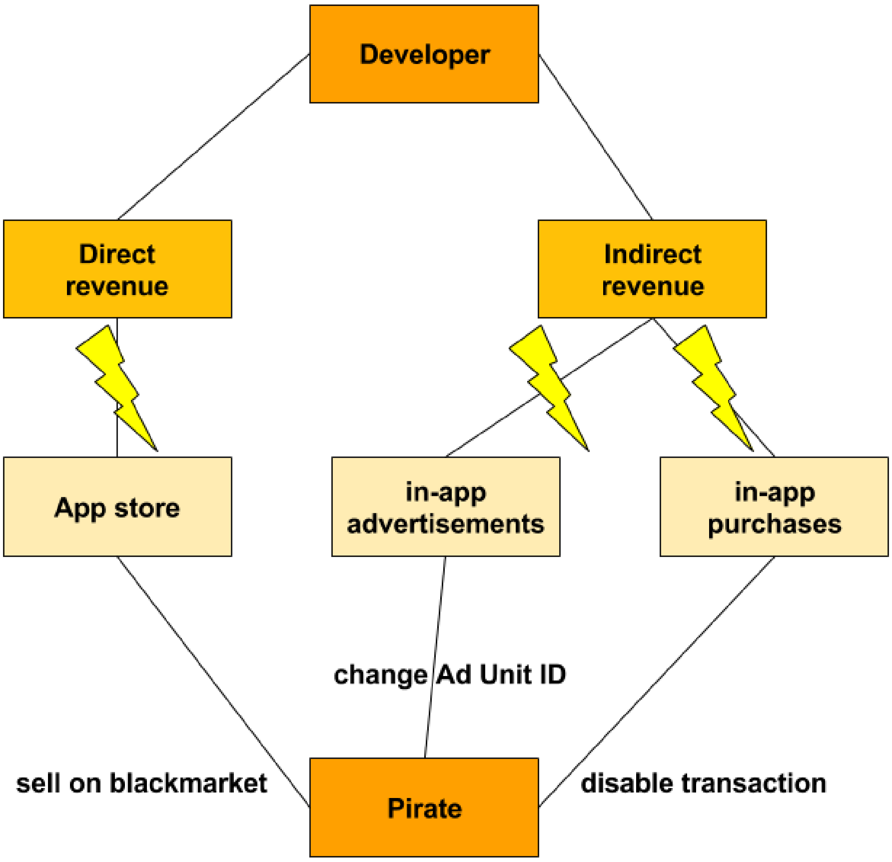


Figure 2.1: Different ways to generate revenue and how the pirate can cut them

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

that they are using a pirated version will connect the unsatisfying behaviour to the developer. This results in the loss of future revenues which are not even connected to this application.
The developer can face also unpredictable scenarios like unforseen traffic since the growth of the application cannot be monitored with tools provided by the marketplace of choice. This can stress the server because they were not scaling accordingly. As revenue from the application is stolen, there may not be enough money for upgrades necessary by legal and illegal use [60].
Developers make a living of their applications. When they do not make a profit from their application, or even lose money with their servers, they can no longer continue. The result is a loss of creativity, ideas and skill for the ecosystem.

2.1.2 Users

The loss of developers in the ecosystem is bad for user, but they can also be harmed by piracy. Users use pirated applications because they seem to be free of charge, but it comes at a price. The application might be altered in different ways, e.g. malware may be included. The user will not notice it right away since these *features* often happen in the background without their knowledge. The application may for instance start using an expensive service, like premium SMS, or upload the contacts to the internet without the user recognizing. Even if there is no malicious content implemented, the application can suffer from bad stability due to manipulated code, like removed license verification, and missing updates when related from an unofficial source. In general, the risk is very high that pirated software has a worse user experience than the original. Pirated software should not be installed since the integrity of the application cannot be ensured without deep inspection. [31] [60]

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 [41]. Sources for pirated applications can be easily found on the internet. A simple search, containing *free apk* and the applications name, returns plenty of results on Google Search. The links direct to black market applications, as Blackmart [29], and websites offering cracked Android Application Package (APK), such as crackApk [38]. They claim to be user friendly because they offer older versions of applications. Their catalog even includes premium apps, which are not free in the Play Store and include license verification mechanisms [27]. Offering these applications is only possible when the license mechanism is removed. They practice professional theft and expose users to threats (see section 2.1.2).

Today Calendar Pro is an example for the dimensions piracy can reach for a single application. The developer stated in a Google+ post that the piracy rate of the application is as high as 85% at the given day. [74] [87] Since it looks like that license mechanisms are no obstacle for pirates and sometimes cracked within days, some developers do not implement any copy protection [54].
Android applications are at an especially high risk for piracy because bytecode in general is an easy target to reverse engineer as shown in the further proceeding.

2.2 Android

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

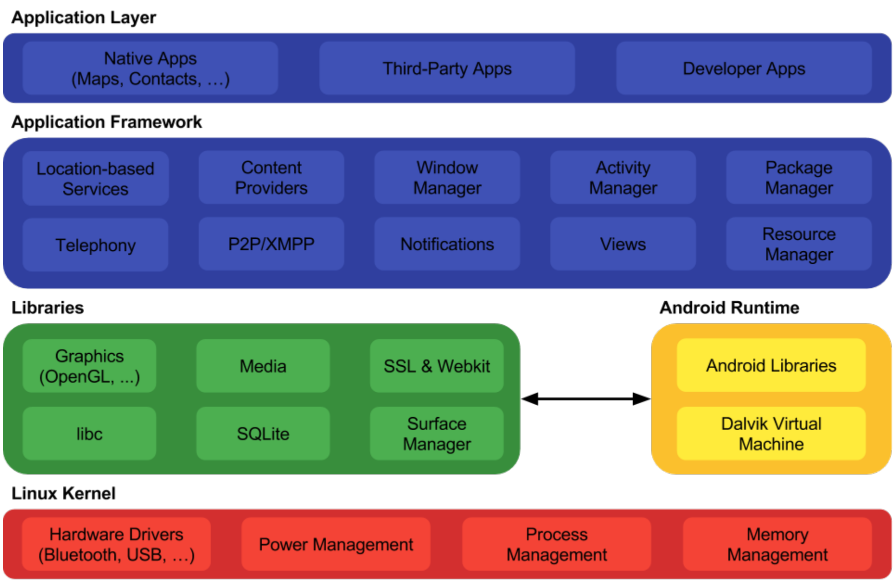


Figure 2.2: Android's architecture [64]

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

agement and controls the device drivers.
The layer above the kernel contains the Android RunTime (ART) as well as the native libraries of the system. ART and its predecessor, the Dalvik Virtual Machine (DVM), will be covered in section 2.2.4 and in section 2.2.5. Usually, Android libraries are written in Java, except the ones which are resource and time critical. They are written in C or C++ in order to boost performance and allow low level interaction between applications and the kernel by using the Java Native Interface (JNI). Examples for native libraries are OpenGL, multimedia playback or the SQLite database.
On top of the libraries and the runtime lies the application framework. This layer provides generic functionality, such as notification support, to applications over Android's Application Programming Interface (API).
The top layer enables the installation and execution of applications.
Using these layers and abstraction allows software to execute standard Unix commands on the kernel and Android to run on a wide range of devices with different hardware and.

2.2.1 Android Application Package (APK)

Android applications are distributed and installed using the APK file format. They can either be obtained from an application store, like Google Play, or downloaded and installed, manually or by using Android Debug Bridge (ADB), from any other source. The APK format is based on the ZIP file archive format and contains the code and resources of the application.
The build process of APK contains several steps which are visualized in figure 2.3.
Since Android applications are usually written in Java, the start is similar to the Java program build process. Android applications are usually written and Java. They have the same build process as standard Java applications. Upon compilation, the source code is compiled to .class files by the Java Compiler javac. Each Java class is stored as bytecode in the corresponding .class file. Java bytecode can be obfuscated, which is topic of section 4.1.4. When all Java classes are compiled to .class files, they are packed into a Java Archive (.jar) file.
Since Android is using a different Virtual Machine (VM) for executing the code, the Java bytecode has to be converted to Dalvik bytecode. The Android Software Development Kit (SDK) provides *dx*, the tool used to convert .class files to a single *classes.dex* file. The VM and the Dalvik EXecutable (.dex) file format will be described in the following. Similar to the Java bytecode, obfuscation can be applied.
The APK itself consits of three parts.

- *classes.dex*, containing the bytecode

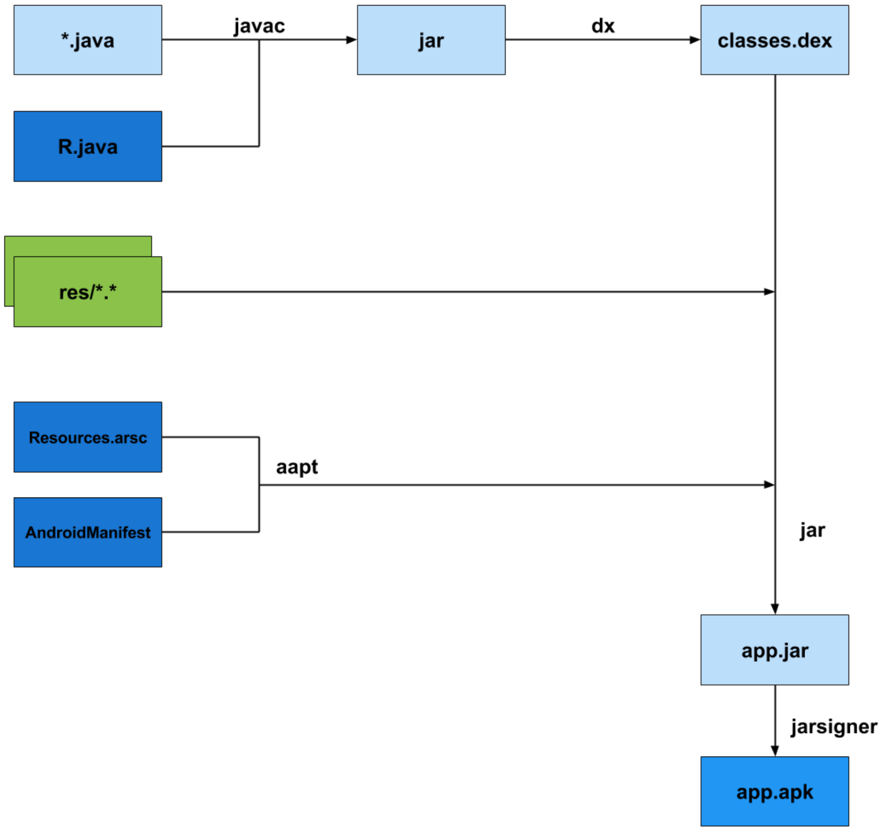


Figure 2.3: APK build process [59]

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

The *apkBuilder* combines these files into one archive file.
Before releasing the application, it has to be signed and zipaligned. The *jarsigner* is used to sign the application with the private key of the developer. It enables Android to identify the developer and support future updates for the application. Afterwards *zipalign* is used to mark uncompressed data. [23] [59]

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

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

Figure 2.4: APK folder structure

configuration data, e.g. the signature [65]. While the static resources, like drawables and layouts, are in the res folder, the resources.arsc contains the compiled resources. In case the application implements native code, it is stored in the libs folder, split by the different processor types, like armeabi-v7a for ARM or x86 for Intel processors. [56] [43]

2.2.2 Dalvik Executionable File Format

As explained in subsection 2.2.1, Android applications are distributed using dex bytecode which is compiled from Java bytecode. Dalvik bytecode is suited to run on the ARM architecture. It supports direct mapping from dex registers to the 32 bit registers of the ARM processor. The instructions are 16 bit mutliples, which makes the dex bytecode less compact than Java bytecode with its 8 bit instructions. The 16 bit instructions result in 218 valid opcodes which have a dest-source ordering for its arguments. [12] In case there are 64 bit values, adjacent registers are used to store it.
Similar to Java bytecode, instructions are not stored inside the method but as a reference pointing to the variable. While in Java each class has its constants, like numbers, strings and identifier names, grouped together in heterogenous pool (see figure figure 2.5, left side), Dalvik bytecode uses one pool for each type. When compiling Java bytecode to Dalvik bytecode, the heterogenous pools of each Java class are merged together in one global pool for each type (see figure figure 2.5, right side). In this process, duplicates in a pool are removed, which reduces memory need for constant but increases the number of references. This is most effective for strings. A decrease of the memory footprint of up to 44% compared to the .jar is possible.

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

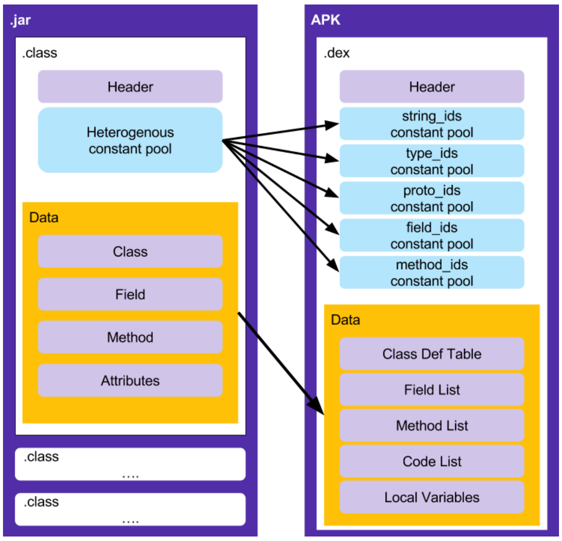
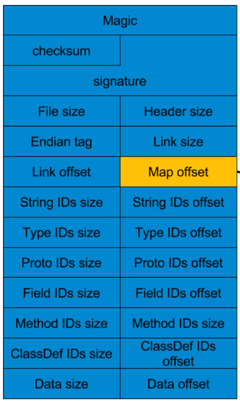


Figure 2.5: .jar to APK transformation [30]

important parts of the header are the checksum and the signature. The checksum contains the Adler32 checksum of the .dex file, including everything except the magic and this field. It is used to detect whether the file is corrupt. The signature contains the SHA-1 signature of the content of the file, except the magic, checksum and this field. The field is used to unqiuely identify the file. When the file is modified, both values have to be updated. [18] [43]

Dex bytecode supports optimization, upon installation improvements for the under-



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

Figure 2.6: .dex file format [59]

lying architecture can be applied to the bytecode. The resulting .dex file is called Optimized Dalvik EXecutable (.odex). The optimization is executed by a program called *dexopt* which is part of the Android platform. The semantics of the two files is the same, but the .odex file has the better performance.
Like Java bytecode, .dex bytecode has a serious flaw. Since bytecode is pretty simple and contains a lot of meta information, decompilation can be succesfully done and the result is easily understandable. At the same time, protection is rarely applied by the developers. This makes these applications an easy target for reverse engineering.

2.2.3 Installing an APK

Before running an application, its APK has to be installed. The installation consists of two major steps. The first step is primarily about verification, while the second step is the bytecode optimization and, in case of ART, the code compilation (see figure 2.7). The differences will be explained in the following subsections.
Before initiating the installation, the APK is checked for a legitimate checksum, the signature is checked and the *classes.dex* structure is validated. In case one action fails to validate, the installation is rejected by the OS.
The installation can be performed in two ways depending on the runtime of the Android OS. In case the DVM, optimisation is applied to the *classes.dex* file and the corresponding .odex file is generated and moved to the Dalvik cache. As a reminder, the .odex is an optimization tailored to the specific architecture of the device in order to achieve the best performance. This is useful due to the the high diversity of Android running hardware and their different processors. This is done once on installation. Future application starts will execute the .odex file instead of the the .dex file. This preprocessed version of the application has an improved startup time. [56]
Currently, the Android runtime of choice is ART. For this runtime, the second step is more complex since the bytecode has to be compiled an additional time. This will be explained closer in section 2.2.5.
After the bytecode is optimized respectively compiled, the application can be run.
When the application is run on the device, Android creates an sandboxed environment for application only. This is achieved by assigning each process an own VM and seperate user ID. This way each application runs separated from the others and has no access on resources except its own. [16]

2.2.4 Dalvik Virtual Machine (DVM)

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

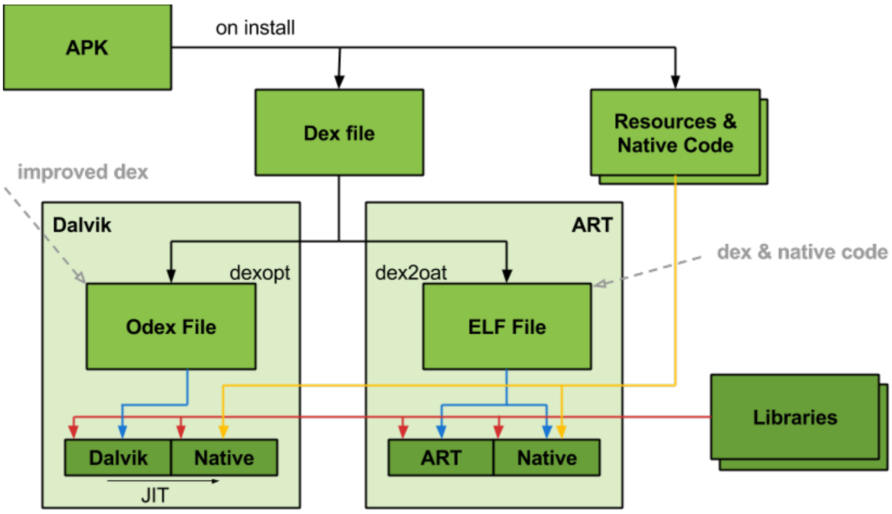


Figure 2.7: Installing an APK on a device [26]

In contrast to a stationary computer, a mobile device has a lot of constraints. Since it is powered by a battery, the processing power and RAM are limited to fit power consumption restraints. In addition to these hardware limitations, Android has some additional requirements, like no swap for the RAM, the need to run on a diverse set of devices and in a sandboxed application runtime. In order to deliver best performance and run efficiently, it has to be designed according to these requirements.
The DVM is a customized and optimized version of the Java Virtual Machine (JVM) and based on Apache Harmony. Even though it is based on Java, it is not fully J2SE or J2ME compatible since it uses 16 bit opcodes and register-based architecture in contrast to the stack-based standard JVM with 8 bit 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 battery driven devices. 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. In addition to the lower level changes, the DVM is optimized for memory sharing. It stores references bitmaps seperated from objects and optimizes application startup by using zygotes. [43] [59]
The last big change made to the DVM was the introduction of Just-In-Time (JIT), which will be part of the discussion in subsection 2.2.5, in Android version 2.2 "Froyo".

2.2.5 Android Runtime (ART)

In Android version 4.4 *Kitkat* Google introduced ART which was optional and only available as a preview through the developer options. ART is designed to address the shortcomings of the DVM.
For backwards compability, ART still works with bytecode in the .dex files format [10]. 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.
Maintaining an VM is expensive, having an interpreter and JIT is not as efficient as native code. Performing JIT each time the application is executed is wasteful. In addition, maintaining background threads require significantly more CPU cycles. Both can be directly translated to slower performance and increased battery usage. The DVM frequently suffers from hangs and jitters caused by the Garbage Collection (GC). With ART Android is following iOS into the 64 bit world, the 32 bit support of the DVM look like a disadvantage, but it is not.
Improvements in ART make the maintenance less expensive, like moving from JIT to Ahead-Of-Time (AOT) and reducing overhead cycles. 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 poorly documented and still chaning a lot. The boot.art file is mapped in memory before the linked .oat file It is mapped to the memory upon zygote startup to provide improved application starting time . In addition to the boot.art file, there are two different .oat files. The boot.oat contains around fourteen of the most used Android framework .jars. The other .oat files are the former .odex files. They are still located in the Dalvik cache, but they are now Extensible Linking Format (ELF) files with the odex file embedded. Instead of *dexopt*, *dex2oat* is used to create these files. [59] [11] [10] [42]
In general, there is still room for improvement since not all code is guaranteed to be compiled. Since the base code is still dex and thus the VM is still 32 bit, ART is not fully 64 bit. The generated code is also not always efficient as from a native compiler but is likely to be improved with LLVM improvements.

2.2.6 Root and Copy Protection

Now that the underlying architecture is portrayed, *rooting* and the original copy protection are explained. *Rooting* or *getting root* is the process of modifying the operation system's software that shipped with a device in order to get complete control over it. The name *root* comes from the Linux OS world where the user *root* has all privileges. This allows to overcome limitations set by carriers and manufacturers, like removing pre-installed applications, extending system functionality or upgrading to custom versions of Android. Manufacturers and carriers do not approve of rooting but they cannot prevent it as the access is usually gained by exploiting vulnerabilities in the system's code or device drivers. Vulnerabilities which can be exploited to gain privileges are quite common. As seen in figure 2.8, there have been over 30 vulnerabilities making it possible gain root rights since Febraury 2015. Details and references of OS vulnerabilities can be accessed on pages like Common Vulnerabilities and Exposures or similar [39] [40].
Today it is easy to exploit these vulnerabilities in order to gain root rights, even for non-techies. There are videos and tutorials available on the internet, even tools to automate the process, like Wugfresh's Rootkits [88]. Rooting is usually bundled with installing a program called *su* which manages the root access for applications requesting it. The exploitation is not without risk since installing bad files can result in the so called *bricking*. Tthe phone is then nonfunctional since the software cannot be executed anymore. [62]
Now that the application is installed and ready to run. Copy protection is applied to prevent unauthorized usage of the app. The downloaded APK, purchased from an application store, is moved to */mnt/app-asec/package.name* folder on the phone. The user has no rights to access the APK in this folder and thus cannot copy it. This mechanism has a major flaw, as copy protection is circumvented when a single user can get hold of the application and redistribute it, e.g. by using root. This mechanism was only an effective measure in the early days of Android when rooting was not easily facilitated. Since rooting voids the copy protection, it is declared as deprecated. All applications are now stored in */data/app/* to which the user has access. Additional ways to protect the applications from piracy have to be applied.

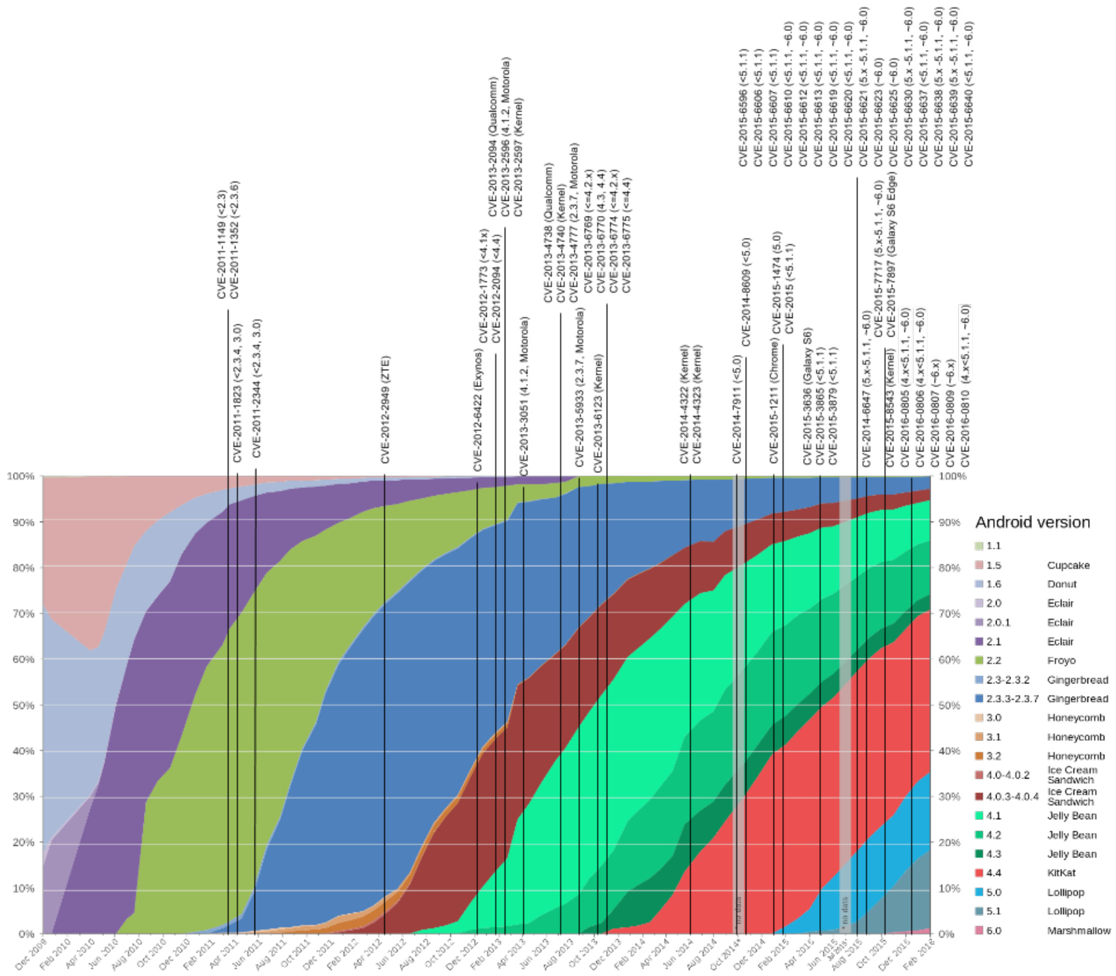


Figure 2.8: Timeline of market share of evolving Android versions and dates of root exploits recorded [44] [1] [39] [40]

2.3 License Verification Libraries

Since the original copy protection of subsection 2.2.6 can be circumvented, a new and secure protection for Android applications is needed. This is not only relevant to Google as the main of contributor to Android and provider of its biggest store. Since Android philosophy is to allow the installation of apps from any source and not only the Google Play, other stores were created to get a piece of Google's Android business. 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 their own ecosystem by selling the *Fire tablets*. They use a flavor of Android tailored to fit Amazon's needs and come at a low price. Samsung pursues a different approach. In addition to a store, they are also offering different services to bind to their ecosystem. There are different Chinese stores as well. They are out of scope here since they have no relevance in the western markets and most of them do not implement any kind of copy protection.
All these stores have to fight the copy protection problem in order make their store attractive and attract developers with low piracy rates.

2.3.1 Google's License Verification Library (LVL)

In order to tackle the copy protection problematic and to give the developer community a possibility to fight piracy, Google introduced the LVL on the 07/27/2010 [35]. It is easy to use and free of charge. The documentation can be found on the Android developers website [20].
Google's approach is based on a network service. It allows to query the trusted Google Play license server in order to determine whether the user has a valid license. Google Play is the name of the former Google market.
The source code for the LVL is provided by Google inside the Android SDK. It has to

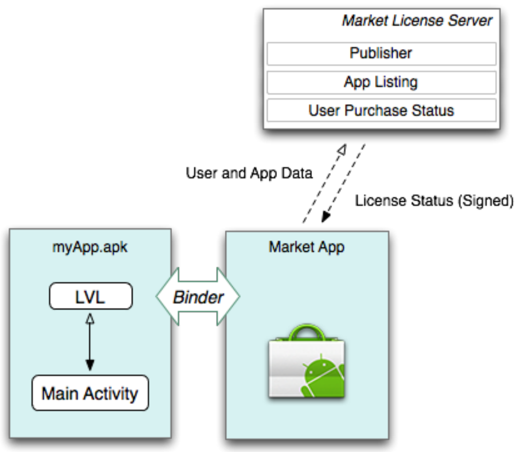


Figure 2.9: Google's implementation of license checking [20]

be manually integrated into the application by the developer. Since the structure and the way it works can be analysed in the source code, it is vulnerable to attacks. For this reason Google instructs the developer to change the library in order to make it unique and less of a target. The LVL can be integrated in only a few steps and does not alter the function of the application.

It is necessary to have a Google Publisher Account in order to take advantage of the LVL. It is used to publish applications on the Google Play Store. The LVL does only work when implemented in an application that is distributed in Google's application store. When the entry for an application is created in the Google Developer Console, an application specific public/private key pair is generated. The use is explained later on. [22]

After an App is registered with the Play Store and the set of keys has been received, LVL can be implemented into the application. The source code of the LVL has to be extracted from the Android SDK and moved to the application project. In order to make use of the library inside an application, three extensions to the application's source code have to be made. [55] [20]

The first extension is the licensing permission in the *AndroidManifest.xml* (see code snippet 2.1). It is necessary for the LVL to work because else an exception will be thrown when the application is run. [22] [14]

The second extension is the callback for the asynchronous handling of the license

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

Code Snippet 2.1: Include permission to check the license in AndroidManifest.xml [14]

verification result. The callback has to cover the possible outcomes, *allow()*, *dontAllow()* and *applicationError()*. The implementation can be seen in code snipped 2.2. The *applicationError()* is used when the license verification cannot be made, e.g. because no internet connection could be established or because the application is not registered with the Google Play server. For each of the methods the developer has to implement the code for how the result should be handled. [20] [22] [14] [55]

The third extension is the license verification call which can be seen in code snippet 2.3. The *LicenseChecker*, responsible for the check, is initiated by passing three arguments. The first argument is the application which is provided by Android. The second argument is the public encryption key. It is retrieved from the Google Developer console and has to be stored in the code by the developer. The third argument is the policy. The policy decides what happens with the response data, e.g. if it should be cached or requested every time. The developer has to define the policy. Google provides two example policies as part of the LVL. Policies require obfuscation to prevent root users from manipulating or reusing the license response data. An example obfuscator is included in the LVL. It is generated using a salt, the package name and the *ANDROID_ID*. The ID is created randomly when the user sets up the device for

```
    private class MyLicenseCheckerCallback implements LicenseCheckerCallback {

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

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

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

Code Snippet 2.2: LVL license check callback

the first time. It is unique and remains the same for the lifetime of the user's device. When everything is provided, the verification is started by passing the callback to the *LicenseChecker*'s *checkAcces()* method. The developer is free to implement the license verification anywhere it is needed. [20] [22] [14] [55]
Upon execution, the information is passed to the Google Play Service client. The

```
        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 2.3: Setting up the LVL license check call

Google Play Service client then adds the primary Google account username and other information and sends the license check request to the server. On the Google Play server, it is checked whether the user has purchased the application and a corresponding response is send back to the Google Play Service client. The response is encrypted to ensure integrity and detect tampering. The Google Play Service client passes it back to

the LVL which decrypts the response, evaluates it and triggers callback accordingly. [20] [22] [14] [55]
The LVL mechanism replaces the old copy protection. It's goal is to provide a simple solution by handling the complicated process, like networking and web services, for the developer. The developer is in full control of what happens with the response and whether access is granted. This license verification can be enforced on all devices which have access to Google Play Store and the Google Play Service. In case the application is installed on a device without the Google Play Service, it cannot bind to it and thus cannot verify the license. Only pirated applications are impacted since the LVL should only be implemented when the application is distributed over Google Play. The actual license check obviously requires connection to the internet, as the LVL needs to connect to the Google server. In case this is not possible, an internal error occurs and the license verification fails. The developer must decide when and how often the license check is done as well as whether the result should be stored for future requests. Depending on this policy, internet connection is needed. [20] [22] [14] [55]

2.3.2 Amazon DRM (Kiwi)

Amazon started its own application store in October 2010 [7] as an alternative go Google Play. The Amazon appstore opened to the public on the 03/22/2011 [8]. It can be used on Android and *Fire* tablets. The store comes with its own Digital Rights Management (DRM) since the Google LVL only works with the Google Play Store. The DRM is called Kiwi as seen in the reverse engineered code in figure 2.10.
The prerequisites for using *Kiwi* are similar to the ones of the LVL, since the developer

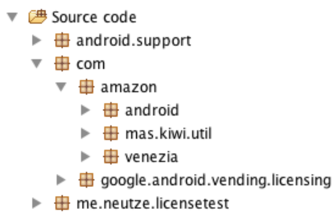


Figure 2.10: Amazon library structure in decompiled application

requires a developer account on the Amazon Developer Service platform. 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]
Amazon has a different approach for implementing the license verification library. Instead of providing the developer the source code, Amazon injects the mechanism automatically into the application when it is uploaded. The developer can chose in

the developer console whether this should be done or not (see figure 2.11). In order to implement the library, the APK is decompiled on the server side, the library is added and the application is compiled again. This requires the package to be signed with a new signature as described in subsection 2.2.1. Instead of using the developer's own key, Amazon uses a developer specific key. The information about the key can be retrieved from the developer platform as seen in figure 2.11. [6]

The implementation can be analysed with reverse engineering. The license verification

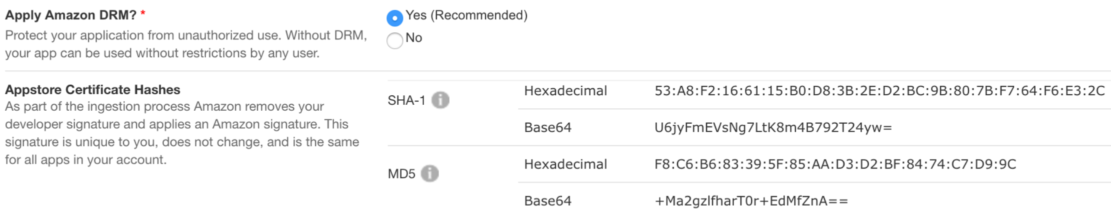


Figure 2.11: Developer preferences in the Amazon developer console [6]

library is wrapped around the original launcher activity of the application. Its logic is not interweaved with application logic. The original *onCreate()* method, which is called when the application is started, is renamed to *onCreateMainActivity()* and a new *onCreate()* is injected. The new method can be seen in code snippet 2.4. When the application is launched, not only the application is initiated as before, but also the *Kiwi* DRM functionality is started by calling *Kiwi.onCreate((Activity) this, true)*.

The license verification works in combination with Amazon's Appstore application

```
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

which acts similar to the Google Play Server. In case Amazon's store is installed on the device, but the user is not signed in, the application prompts the user to sign in. Since signing in requires an connection to the internet , *Kiwi* indirectly depending on it as well. It is different when the wrong user is signed in or the store is not even installed. In this case, the application shows a warning that the app is not owned by the current user, respectively that the Amazon Appstore is required and cannot be found.

2.3.3 Samsung DRM (Zirconia)

Another major player in the smartphone business is Samsung [36]. With *GalaxyApps*, renamed from *SamsungApps* in July 2015, they offer an application store to their Android devices. Application distributed in that store can be protected using *Zirconia* [71].
The way the library works is similar to the LVL. The library queries the Samsung server to verify the license of the user in order to prevent unauthorized usage of the application. The library can be downloaded from Samsung in an archive file [71]. It contains the compiled Zirconia library as a .jar and additional native libraries. The integration requires both file types to be added to the application.
The implementation in the application code is done the same way as in the LVL. The developer is free where to implement the three code additions needed.
First of all the required permissions have to added to the *AndroidManifest.xml*. Zirconia needs access to the internet and to the phone state (see code snippet 2.5).
The second addition is the implemetation of the *LicenseCheckListener*. It contains the two

```
...
<uses−permission android:name="android.permission.INTERNET" />
<uses−permission android:name="android.permission.READ_PHONE_STATE" />
...
```

Code Snippet 2.5: Include permission in theAndroidManifest.xml [71]

results, either valid or invalid license verification result. While *licenseCheckedAsValid()* contains the code for success, *licenseCheckedAsInvalid()* is used when the license cannot be validated. . The third addition is initialization of the license check. Zirconia handles all everything in its own. The developer just has to set the listener for the result and start the check by calling the *checkLicense()* method.
Zirconia always follows the same internal pattern when the license check is executed. First, it is queried for a stored license. If a stored license exists and it is valid, the check passes and no internet connection is required. Otherwise Zirconia sends information of the device and the application to the server. The server evaluates whether the user is authorized to use the application and replies accordingly. The response is unique for each device and application combination and thus cannot be used on another device. In case the access is granted, Zirconia stores the license on the device. The next time the license check is initiated, the same flow is done.

```java
        @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

```java
        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 2.7: Setting up the Zirconia license check call

2.3.4 Conclusion

All license verification libraries are working inside the application and query a trusted source. They do not prevent redistribution or copying but enforce the authorization when the application is run. The result of the verification is always binary, either access is granted or it is prohibited. The libraries can be abstracted as in figure 2.12.

2.4 Code Analysis

Cracking tool have to modify the application code since it contains the license verification. The investigations starts with analysing the APK's *classes.dex*. The goal is to identify how the circumventing of the license verification mechanism is achieved. This aquired knowledge is later used to find countermeasures for the crackign tools.

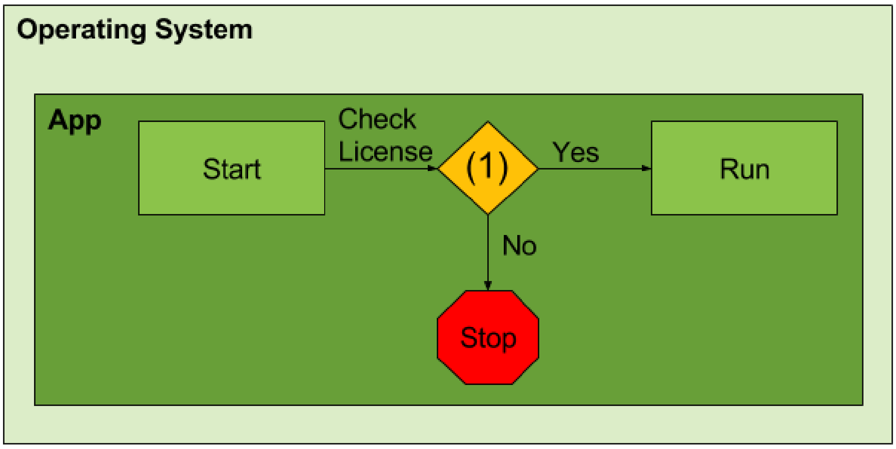


Figure 2.12: Abstraction of the current license verification mechanism. The library is represented by (1)

The reengineering has to be done by using different layers of abstraction. While dex and smali code reflects local changes, Java code is used to detect functional changes. It is possible to access the applications Java code since the dex code can be decompiled as seen in figure 2.13. This requires different tools which will be explained in this section.

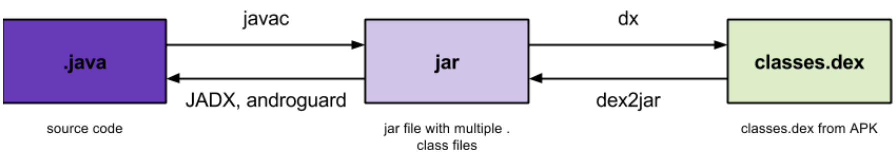


Figure 2.13: Java .class and .dex can be transformed bidirectional [59]

2.4.1 Retrieving an APK

The analysis is performed on a computer since most tools are available on that platform. In order to analyse the application, its APK has to be extracted from the phone and transfered to the computer.
Most APKs are stored in */data/app*. The filesystem of an Android device be accesses using the *adb shell*, which is part of the Android SDK. The user has no read rights on the folder and thus cannot see the content of the folder. In order to pull the desired application package the exact name is required. The package manager can be used

to list all installed applications and their package names. The list can be retrieved using *pm list packages -f* in the *adb shell*. In case the phone is *rooted*, the user can list the content of the server in the *adb shell*. The format of the application package name is <*namespace*>.<*appName*> and contains the application as *base.apk*. The APK can be extracted to the current folder using *adb pull /data/app/<namespace>.<appName>/base.apk* on the computer's console.

2.4.2 dex Analysis

The *classes.dex* contains the application code and has to be modified by the cracking tool to carry out the attack. It makes it possible to point out which bytecodes have been modified. This is the reason to analyse the dex bytecode and use it as the first abstraction layer.
The extraction of the *classes.dex* is done using a simple script shown in code snippet 2.8. The APK is an archive file and can be unpacked using *unzip*. The content is unpacked

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

Code Snippet 2.8: Script to extract the .dex bytecode from the APK

to the destination which is added with the parameter *-d location* as seen in line 3. The extracted *classes.dex* is still formatted in binary. Hexdump is used to convert it to a hexadecimal view. In this view, one character is 4 bit, thus one tuple is one byte and two bytes form a 16 bit opcode. This presentation allows to identify opcodes and translate them using an opcode table [66].
The output contains the line number, the bytecode and the ASCII translation. Code snippet 2.9 is an example of the beginning of the *classes.dex* file. The first 8 byte or 16 hex tuples, *64 65 78 0A 30 33 35 00*, are the .dex file magic, which identifies the file type. Translated to ASCII, the result is *dex.035..*

2.4.3 Smali Analysis

The smali presentation of the bytecode is the second abstraction layer. Smali is used to disassemble the dex bytecode into human friendly mnemonics. It makes it easier to interpret the result of the changes in the bytecode regarding functionality and location. The disassembling from dex to smali format is done using *baksmali* [50]. Assembling and disassembling of dex and smali is possible without the loss information since they

```
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 2.9: Hexadecimal view of classes.dex as classes.txt

have a bijective mapping[50]. It takes the APK and disassembles *classes.tex* file. The output is a file for each class, containing the smali code. The resulting syntax is loosely based on Jasmin's syntax This process is done using the script in code snippet 2.10. An example of a smali view can be seen in code snippet **??**. It is easier to understand

```
#!/bin/bash
#baksmali
java -jar baksmali.jar -x base.apk -o /smali/
```

Code Snippet 2.10: Script to generate the corresponding smali code for a given APK

than the dex presentation. The advantage over the dex bytecode is that the structure of classes and methods is restored which makes it easier to analyse without additional tools. In addition, the content of variables, as in line 3, can be identified without big effort. Bytecode opcodes are replaced with their actual opcode name. This enables the reader analyse the application's work flow similar to the source code.

```
# virtual methods
.method public magic()V
  const−string v4, "android_id"
  ...
  move−result v0
  if−eqz v0, :cond_7
  ...
.end method
```

Code Snippet 2.11: smali code example

2.4.4 Java Analysis

The third abstraction is the decompiled Java code. The goal is to reverse engineer a result very close to the source code including the changes. It suits best for the analysis since it contains the most information, such as variable and method names. The representation is close to how the developer implemented the application.
As seen in figure 2.13, the .dex files are isomorphic to the corresponding Java .class files. This makes it possible to decompile the dex bytecode into Java code. The decompilation to the exact source code cannot be achieved since in the compilation process some information is lost. Another problem is the otimization of the dex bytecode for mobile usage. The specific mobile patterns are unknown to the Java decompiler. The outcome is not always sufficient and for this reason two different decompilers, DAD and JADX, are used.
The first decompiler is DAD, short name of "DAD is A Decompiler". It is part of Androguard [9]. It works with the dex bytecode and does not require third aprty tools like dex2jar [67]. Code snippet 2.12 shows how it is used to decompile the APK into Java code.
The second decompiler is JADX [75]. It supports decompilation for dex to Java code. It

```
#!/bin/bash
#androguard
python androdd.py -i base.apk -o /java/dad/
```

Code Snippet 2.12: Script to decompile to Java using androguard

can be used in the command line as seen in Code snippet 2.13.

```
#!/bin/bash
#jadx
jadx -d /java/jadx/ --deobf --show-bad-code base.apk
```

Code Snippet 2.13: Script to decompile to Java using JADX

2.4.5 Detect Code Manipulations

The *classes.dex* and its abstractions contain a lot of code. The best way to recover the changes is to use diff. Diff is a standard command line tool which is used to compare two files and in order to recieve the difference between them. It is applied to the different code abstraction of the original APK and the attacked APK.

It returns the location as well as the orignal and changed code. The automatic discovery

```
#!/bin/bash
#dex
diff -r /dex/original/ /dex/manipulated/ > dex.diff
#smali
diff -r /smali/original/ /smali/manipulated/ > smali.diff
#dad
diff -r /java/dad/original/ /java/dad/jadx/ > dad.diff
#jadx
diff -r /java/dad/original/ /java/dad/manipulated/ > jadx.diff
```

Code Snippet 2.14: Script to compare the original and manipulated APK to see the modifications in the different presentations

and listing of the changes saves a lot of time. The example diff of a dex file is presented is code snippet 3.1.

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

Code Snippet 2.15: Diff on Dex level for N1 pattern

3 Cracking Android Applications with Lucky Patcher

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

3.1 Lucky Patcher

On the official website, Lucky Patcher is described as "[...] a great Android tool to remove ads, modify apps permissions, backup and restore apps, bypass premium applications license verification, and more" [33]. It is written by a developer called ChelpuS and currently on version is 6.0.4 (on 02/17/2016).
Lucky Patcher offers the removing of the licensing in premium apps to crack their DRM, to remove in app ads, change and restrict permissions and activities as well as to create modified after applying one of the feature above on the original APKs [33]. Since copy protection by Google is deprecated, all applications are stored in */data/app/*. The user as well as Lucky Patcher can access this folder and copy applications from it. *Root* and busybox, an application which provides standard UNIX tools for Android[82], is only required when the application should be modified on the device, which is explained later. Lucky Patcher requires no technical knowledge and offers automatic cracking for non professionals. This combination makes it a popular and an effective tool with a high damage potential. [63]
This thesis focuses on how Lucky Patcher is bypassing the license verification mechanism of applications. The goal of circumventing the license check is to make the pirated application work as if had been legally acquired in the store. As described in section 2.3, the license verification is implemented as client-server connection. The app sends information about the user and the application to the server, which checks and verifies the given information, and replies with an approval or rejection. The server communication is secure against man in the middle attacks or spoofing, as messages are private key encrypted [63]. Lucky Patcher is taking a different path by modifying

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

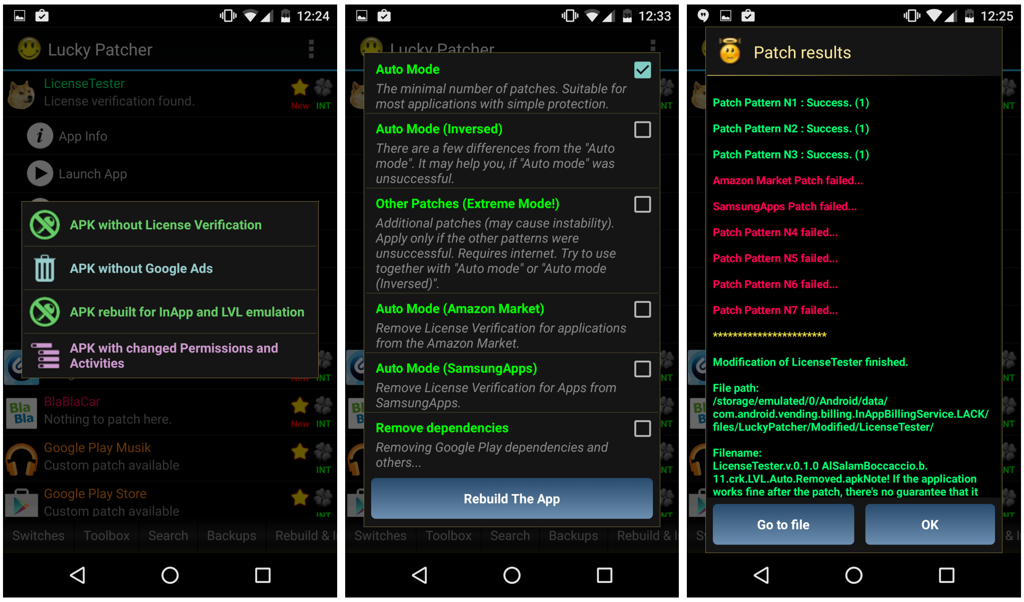


Figure 3.1: Left to right: Features offered LuckyPatcher, modes to crack license verification and the result after patching

Using Lucky Patcher is fairly simple. The application can be downloaded from the official website [33] as an APK and installed on the device. In order to have the features available, the device has to be rooted. On startup of Lucky Patcher, all installed applications are shown in a list. An info text about what patches can be applied is shown below each name. When a the application is selected, a submenu offers various actions, e.g. to get information about the app or run it. Figure 3.1 shows the menu of patches which can be applied to the application. These patches can either be applied in two ways. Choice one is by applying the patch directly on the device and should be taken when *root* is available. This method creates an .odex in the Dalvik cache on the phone. Choice two is by creating a modified APK, which does not require *root*. In order to to run the cracked application, the original APK has to be removed and replaced by the modified one. Upon selecting one of the two choices for removing the license verification, different patching modes can be selected as seen in figure 3.1 in the middle. Their description is rather short and does not offer information of how the modes are working. After selecting the mode, the patching is done and the result is shown. As seen in figure 3.1 on the right, different patching patterns are applied

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

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

3.2 Code Analysis

The code analysis was done using Lucky Patcher in version 6.0.4 using the two tools (Androguard and JADX) described in subsection 2.4.4. The reversed code was inspected using a text editor like Atom [46].
Before analysing the code, a look is taken at the structure. When loading the folder of the reengineered code into the editor a lot of different code folders can be spotted. These code folders hold the packages of Lucky Patcher itself. libraries used and resources. The libraries can be divided into four categories.

1. Android Support Library v4 with many of it's modules. It is used for downward compatibility of Android related functions of Lucky Patcher itself.
2. Lucky Patcher code. They are located in two places. Utility functions, like "copy file" and "rename", are stored in the package com.chelpus utility functions The *application code* itself can be found in the package *com.android.vending.billing.InAppBillingService.LACK*. It contains the activities and functions which are used for cracking applications.
3. The third category are support libraries required by Lucky Patcher to apply it's cracking mechanisms. This includes libraries, e.g. axml [34] for serializing the AndroidManifest.xml from Android binary into an ASCII formatted, human readable xml and zip4j [61], a Java library to handle ZIP files. The fourth and last category is a modified billing and license library. It is applied in combination with a proxy to redirect inapp billing and licensing calls.
4. Already cracked LVL.

Besides the four code folders, the asset folde stores different predefined custom patches which can be applied applications [63].
The code itself

3.3 Analysis of Patched Applications

Since the analysis of the code is hard and does not uncover its functionality voluntarily, an additional analysis of the reverse engineered code of the outcome of Lucky Patcher is done. The analysis is done to identify the parts of the code manipulated by Lucky Patcher and later on find solutions to protect these weak points. Different applications are used to get a variety of results and see how different implementations are attacked. They are patched and output as modified APK by Lucky Patcher as described in section 3.1. Since the code itself is modified, a static analysis is sufficient. Afterwards they are analysed according to the methodology shown in section 2.4. The analysis is done with modified APKs and not on the device directly since the resulting .odex files are device specific and cannot be used for a general conclusion.
The goal of reverse engineering the code and comparing it to the original application is to see the changes on different levels. This includes the .dex level, on which Lucky Patcher works, the smali level, which makes the .dex code human readable friendly, as well as Java level, on which the functionality 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.
In order to have a reference application with known source code and an implementation according to the tutorial of LVL [14], a test application is created. This application is called *LicenseTest*. It makes it possible to analyse how Lucky Patcher works on the most basic version. Besides circumventing the Google LVL, Lucky Patcher supports the cracking applications from Amazon and Samsung as well. For Amazon's Kiwi DRM, the same application, with deactivated LVL, is uploaded to Amazon and injected with their DRM. The analysis for Samsung's Zirconia DRM is only done by using *LicenseTest* with deactivated LVL as well, since the library is implemented the same way into all applications. This can be assumed since the library is included as a .jar and cannot be modified. In addition to the basic application, other applications were analysed as well,to see how Lucky Patcher handles different implementations. The applications, are Runtastic Pro[70], version 6.3, and Teamspeak 3[83], version 3.0.20.2, for the LVL and A Better Camera [5], version 3.35, for the Amazon DRM. These apps were chosen since they were already owned and approved to be included into the thesis by the developers. They include Google's LVL and Amazon DRM.
In addition to the code analysis, the modified application is installed on different devices to evaluate the success of the crack. This is possible since the modified application can be installed on any device. The goal is to identify whether the crack works even though the corresponding store, root or internet connection are not available.
As described before, Lucky Patcher offers different modes to patch applications. Each mode uses a set of patterns which each change a piece of binary. These patterns are

shown in figure 3.1 on the right. A pattern is a set of predefined sequences of bytecode in which a certain values are modified. In order to discover applied patterns and to evaluate each mode, each mode is applied on each application.
These are the different modes and what Lucky Patcher describes them as.

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

3.4 Patching Patterns

In order to identify the structure of the single patterns, the code of the original code was compared to the cracked output. The changes in the code were inspected on dex, smali and Java level with the tools explained in Section 2.4. After analysing same patterns in the different modes it was identified that these patterns can be summed up as one.
The names of the patterns are taken from the patching result output in figure 3.1 on the right. The number next to the pattern indicates how often it was applied to the application. This number is not always correct. The patterns Nx are used to circumvent the LVL while the Amazon and Samsung patterns are tailored to do the same with the library of the respective store. Additional knowledge on the patterns was was gained in the analysis.
Before explaining the patterns in detail, some information has to be provided. When analyzing .dex files, instead of using hexadecimal values like *0x0a*, a simplified presentation as *0a* for improved overview in the diff files is chosen. The opcodes used by the DVM are taken from on of the documentations [66]. When converting .dex files to smali files, the arguments of the opcodes are transfered to variables, e.g. x in dex code is vx in smali.
Since changing the dex code of an application results in a new checksum, the code has

to be resigned as well. This changes can be seen in the diff of the dex files. They are not explicit mentioned in the analysis since it is no direct change of the attack.

Patch Pattern N1

Pattern N1 is present in all patching modes except the solo extreme mode. It tagets the *verify()* method of *LicenseValidator* class in the *com/google/android/vending/licensing/* folder. This method is responsible for dectypting and verifying the response from the license server [21].
In can be seen in the dex code in code snippet 3.1 that the the blocks *1a* and *0f* are swapped in their order.

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

Code Snippet 3.1: Diff on Dex level for N1 pattern

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

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

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

```
- case NOT_LICENSED: handleError(); break;
+ case NOT_LICENSED: handleResponse(); break;
+ case LICENSED_OLD_KEY: handleError(); break;
```

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

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

Patch Pattern N2

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

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

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

Code Snippet 3.4: Diff on Dex level for N2 pattern

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

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

Code Snippet 3.5: Diff on Smali level for n2 pattern

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

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

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

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

Patch Pattern N3

Pattern N3 is different than the other patterns since there are two versions of it. The first version, N3, is used in auto mode while pattern N3i is used in the inversed auto mode. The name N3i is chosen since it is used in the inversed mode. LuckyPatcher does not make a difference between them in the result ouput. They are combined under the same number since both attack the same logic inside the classes defining the policies. In case of the basic implementation of the LVL, these classes are the *APKExpansionPolicy* and *ServerManagedPolicy* in the *com/google/android/vending/licensing/* folder. Those two classes are examples of policies offered by Google [21]. Pattern N3 attacks their *allowAccess* method.
In case of pattern N3, *01* is replaced with *11*, while in case of N3i *11* is replaced by *01* (see code snippet 3.7).

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

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

Code Snippet 3.7: Diff on Dex level for N3 pattern

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

```
@@ Pattern N3 @@
```

```
- const/4 v1, 0x0
+ const/4 v1, 0x1

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

Code Snippet 3.8: Diff on Smali level for N3 pattern

The resulting code is shown in code snippet 3.9. The pattern attacks the result of the function. While N3 is used to attack code where the default return value is *false*, pattern N3i is used when the default value is *true*.
Both patterns attack the class's *allowAccess()* method which evaluates whether the

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

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

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

verification result is according to the policy or not. The return value is only changed in case the test of the input is evaluated accordingly, if this is not the case, it is not changed. The pattern changes the default return value and thus makes the result of the evaluation negligible since the desired result is already set. There are two versions of the pattern to counter inversed logic which can be inplemented easily.

Patch Pattern N4

Pattern N4 was only applied once in the test sample and is part of the auto and auto inverse patching modes. The target of the pattern is the *LicenseChecker* class of the LVL which is responsible for initializing the license check in its *checkAccess()* method [21]. As seen in code snippet 3.10, it replaces *38* with *33*.

```
@@ Pattern N4 @@
```

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

Code Snippet 3.10: Diff on Dex level for N4 patch

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

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

Code Snippet 3.11: Diff on Smali level for N4 patch

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

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

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

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

Patch Pattern N5

As part of the extreme mode, pattern N5 works on the .dex file similar to pattern N2 on the *LicenseValidator*'s *verify()* method. It does not affect the standard implementation of the LVL.
In the Java diff in code snippet codeSnippet:n5DiffJava the result of patching can be seen. Instead of setting the variable to the result of parsing first value of the array *v2_1*, the value of the array is parsed but the variable is set to *0*.

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

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

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

Patch Pattern N6

Pattern N6 is part of the extreme mode and, similar to the pattern N1, N2 and N5, it attacks the *verify()* method in the LVL's *LicenseValidator* class.
This pattern changes three values of the .dex file which can be seen in code snippet 3.14. The first changed value is *38* which is replaced by *12*. The second value is *06* which is replaced by *00*. The third change is the replacing of *4a* by *00*.

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

Code Snippet 3.14: Diff on Dex level for N6 patch

The change of the code structure is more visible in the code snippet 3.15. The first results in the initialization of *p2* with *0*. The second change is required since moving a value to a variable only requires two arguments instead of one argument plus the destination offset. In order to get a valid syntax, the code has to be changed accordingly. The result is interpreted by the decompiler as a *nop* operator and an empty line. The third change is the replacing arguments *p2* and *v4* of the *if-eq* evaluation with *v0* for each.

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

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

Code Snippet 3.15: Diff on Smali level for N6 patch

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

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

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

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

Patch Pattern N7

The final pattern for the LVL is pattern N7. It is applied in the extreme mode and as this mode indicates, it takes a harsh approach on applying itself. It does not only patch the *ILicenseResultListener* class's *onTransact()* method, which is the implementation for the callback for interprocess communication and receives the async response from the license server [21]. In addition the pattern is applied to all classes eligable in the `com/android/`. It can be described as the bruteforce version of pattern N2 and thus should only applied when the other modes are not successfully since the outcome might not be stable anymore.
Similar to pattern N2, pattern N7 attacks by initializing the corresponding variable with *false* instead of moving a result of a method into it.

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

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

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

Overview for Patching the LVL

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

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

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

Table 3.1: Patching patterns applied by each mode

Amazon Market Patch

Amazon does not have different pattern. Since the Kiwi library is injected by Amazon and cannot be customized by the developer, only one pattern is necessary. The pattern is applied twice. The first class tageted is *com/amazon/android/licensing/b.java* while the second class is *com/amazon/android/o/d.java*. Since the injected library is obfuscated, no meaningful names for the methods can be presented.
The Amazon pattern works similar to the LVL's pattern N4 and replaces (*38*) with (*33*).

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

Code Snippet 3.18: Diff on Dex level for Amazon patch

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

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

Code Snippet 3.19: Diff on Smali level for Amazon patch

The Java code makes it easier to identify the functional changes of the attack Since the if evaluation is always wrong inside the *b.java* class, the case when the response code is not *LICENSED* is never called. The same 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 returns true. After patching, the result is always true since the null check is always false as in *b.java*.

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

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

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

Samsung Market Patch

Similiar to the Amazon's Kiwi library, Samsung's Zirconia library cannot be modified as well. Thus cracking the library only requires one patch as well. The patch is applied on the *LicenseRetriever* and *Zirconia* class in the *com/samsung/zirconia* package. Unlike Amazon's Kiwi DRM, Zirconia is not obfuscated and thus better to understand. The Samsung patch uses two patterns, called S1 and S2 in order to distinguish between them. While S1 is applied on both classes, S2 is applied twice but only on the *Zirconia* class.
While Pattern S1 replaces *d6* with *00*, pattern S2 uses *12* instead of *0a*.

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

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

Code Snippet 3.21: Diff on Dex level for Samsung patch

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

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

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

Code Snippet 3.22: Diff on Smali level for Samsung patch

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

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

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

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

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

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

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

3.5 Conclusion and Learnings

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

	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 [63]

UNARY KOMMT HIER VOR

Es muss generell immer abgewogen werden zwischen Reichweite und Sicherheit. Von Output den Lucky Patcher gibt, sind die auto patching modes für Google, Amazon und Samsung, die großen Player. Ein Developer muss seine App dort anbieten um Aufmerksamkeit zu bekommen. Deswegen sind diese Stores auch so gut "maintained"

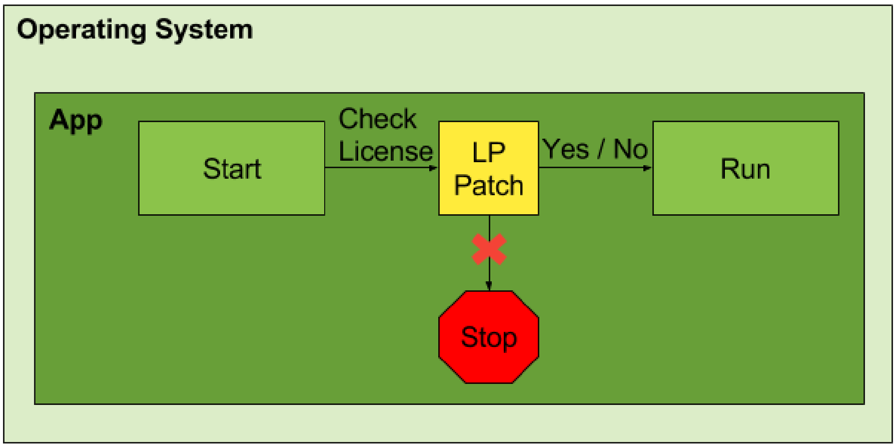


Figure 3.2: Abstraction of the current attack on the license verification mechanism

von Lucky Patcher. Im Falle, dass ein Developer "Sicherheit" vorzieht und seine App in einem alternativen Store anbietet, gibt es zwei Scenarien. Entweder entwickelt jemand einen Custom Patch (dex oder native Angriff) wenn ein "allgemeines Interesse" besteht oder die App ist uninteressant und erhält keine Aufmerksamkeit, weder von LP noch Kunden.

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

unary outcome beschreiben

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

4 Countermeasures for Developers

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

4.1 Extend Current Library

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

4.1.1 Modifications on the Google LVL

Attacking with Lucky Patcher is often successful since many developers do not customize the LVL. One reason for this is developers using the library only against casual piracy. It prevents users, who try to copy APKs directly from one device to another, from using the application. Another reason is that they do not know where exactly to fortify the library and thus they do not want to spend additional work. [53] [63] This thesis presents two approaches for making attacks on the library less succesful.
The first approach is to actively go against Lucky Patcher's patterns by modifying the identified parts of code.
The second approach is to implement the LVL with native code which cannot be targeted by patterns.

Modify the Library

Lucky Patcher's patching is reliant on applying its patterns. Patching application code is the most effective and most common attack to modify the license logic. Going actively against these patterns should always be the first step to challenge Lucky Patcher. This can only be applied to the LVL since it is the only library where the source code can be accessed by the developer. For Amazon and Samsung, this propose can be useful as well but it is more difficult to implement since these two libraries have only one entry and exit point in the applicaiton.
Modifying and improving the LVL does not only protect from patterns. Increasing complexity of the application's bytecode makes it unqiue and harder to reengineer. [53] There are three areas the developer should focus on when modifying the LVL [53].

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

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

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

These are only examples and creativity is welcome since the resulting implementation should be unique. [53]
The entry and exit points can be attacked by creating a counterfeit version of the LVL that implements same interface. An unquite implementation provides resilience against this attack. It can be achieved by adding additional arguments to the *LicenseChecker* constructor as well as the *allow()* and the *dontAllow()* methods. [53]
Attackers do not only target the LVL but the handling of the result in the application as well. This can be prevented by handling the mechanism in a separate activity. In the original activity *finish()* will be called and the attacker will be stuck in the new activity. This prevents from scenarios where the attacker voids methods which would prevent further proceeding. In addition the license verification can be postponed to a later point in time since attackers are expecting it on the the applications launch. [53]
These modifications are easy to apply and make the implementation unique. The unlimited ways to implement make it hard to attack automatically. This does not ensure total protection against Lucky Patcher since every application is patchable in some way. A determined attacker, willing to invest a lot of time and work in disassembling and reassembling, can eventually hack the implementation. The aim of the developer is to make the work of the attacker as hard as possible to the point where the profit is not worth the time. [53]

4.1.2 Native Implementation of LVL

Lucky Patcher's automatic patching modes, as described in chapter 3, target the application's .dex file. Since Android supports the Native Development Kit (NDK), it is possible to implement parts of the application using native code.
Usually the NDK targets CPU intensive 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 [15]. In case of the LVL this is a desired side effect. Native code, in opposite of byte-code, does not contain a lot of meta-data, such as local variable types, class structure. It information discared when the code is compiled and thus makes it harder to understand the code.
There are two scenarios for creating a native implementation of the LVL.

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

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

and safety. The attackers have to invest time to analyse the native code of the license verification and the implementation into the application itself. Only after reverse engineering the native code, they can start with searching for a way to break the license verification mechanism, repack the native code and make it available as a custom patch for Lucky Patcher. This scares off attackers since the circumventing of the license verification requires a lot of knowledge and time. As long as the attacker is evaluating the workload with the gain of cracking the application, it is considered a safe method, implied the developer has enough available resouces. [63]
Instead of providing Java code, Google could provide a native version of the LVL. In the beginnign, it would be harder to find vulnerabilities than it was with the Java version. It would take some time for the attackers to crack the library but it would be justified for the attackers since the library would be implemented into all Play Store applications. Now this license verification would face the same probles as Amazon's or Samsung's libraries since one custom patch applied by Lucky Patcher would be able to crack all applications. For this reason, the implementation should include two essential features.

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

In addition to making the license check native, parts of the application should be moved into the native code. This protects against attacks where the call of the license verification library is skipped.
In general, the proposal is simple, but the implementation is much harder. Until now, no big companie has come up with such approach. This indicates that still a lot of reseach and work has to be done to implement this solution. At least, first steps in this direction have be made by addressing dex and its vulnerabilities. [63]
As long as the the license verification library is implemented in native code, the automatic patching modes of Lucky Patcher do not work. Attackers have to analyse the native code and create a custom patch for the application. These custom patches can already be found for some applications in form of modified *.so* files.

4.1.3 Tampering Protection

The correct execution can only be guaranteed as long as the integrity of the environment is ensured. When circumventing the LVL the code has to be modified. Unless done precisely, this can be detected, e.g. Lucky Patcher does not have the developers signature and thus uses a different one. Other breaches of integrity are debuggability and *root*, cracking applications and unauthorized installation. In order to be able to detect this and prevent the execution of the application a priori, additional checks are implemented as seen in figure 4.1. Since all tampering countermeasures have the same

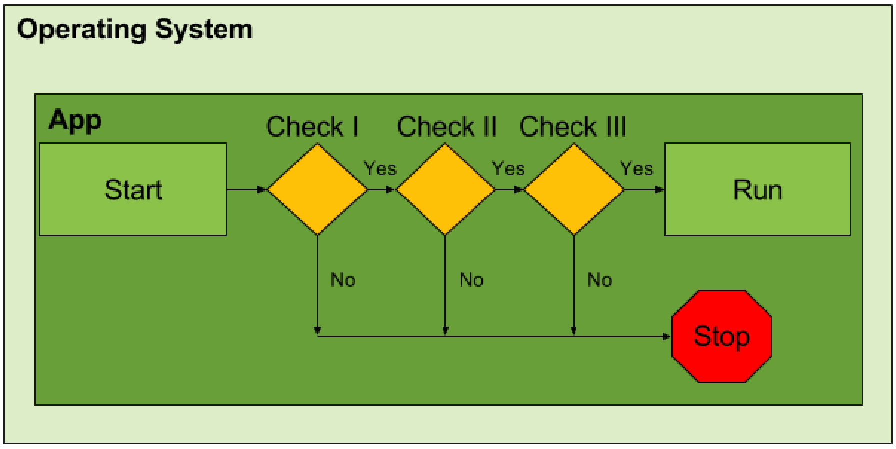


Figure 4.1: Introduction of additional tests to check environment and integrity of the application

pattern in operation, they can be circumvented easily. The goal of these checks is to increase the workload of an attacker since the code has to be analysed in order to find, understand and patch them. In order to make this task even more time consuming, these checks can be obfuscated and spread inside the application. Even the possibility of implementing them in native code can be considered.
This does not prevent Lucky Patcher itself from working, but it offers an additional layer of security which has to be voided.

Debuggability

Enabling debugging allows the developer to use aditional features for analysing the app at runtime, like printing out logs [19]. 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

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

Code Snippet 4.1: Example code for checking for debuggability

can be 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 [57], which is said to be used in security critical applications like Android Pay [17] [69].
Since root is achieved by moving the *su* file to the file system, the application can search in the common locations for it. In case the search is successful, the execution of the application can be terminated.

Lucky Patcher

Having Lucky Patcher installed is an indicator that the user is attacking license verifications libraries. The check is not as strict as the one for root and 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 [89]. Some custom Android versions already include a library called *AntiPiracySupport* [37] 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 package is installed. In case information is available and thus the application is installed, the check stops the application.

```
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

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. Additional stores can and should be added in case the developer decides to offer the application in another store.
This feature should be implemented with caution since Google notes that this method relies on the *getInstallerPackageName* which is neither documented nor supported and only works by accident [53].

```java
    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: Example code for checking whether Lucky Patcher is installed on the device

4.1.4 Obfuscation

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

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

man versucht sich vor neuen angriffen zu schützen indem man reengineering zeitintensiver macht developer can make hackers task immensely more difficult to the point where it is not worth the time[53] reengineering cannot be vermiede[63] best is to apply technqiues to make it as hard a possible[63] obfuscate to make it difficult tor everse engineer[53] it is not possible to 100 percent evade reengineering, but adding different methods to hide from plain sight of reengineering tools[63]

```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: Example code for checking the origin of the installation

Reverse engineering and code protection are processes which are opposing each other, neither classified as good nor bad[56] "good" developer: malware detection and IP protection[56] "bad" developer: analysis for attack and analysis resistance[56] [56]

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

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

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

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

number of commercial and open-source obfuscators available for Java that will work with Android[53] a few dex obfuscators exist, with different approaches[59] 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)[59] can add dead code and dummy lopps (minor impact of performance)[59] can also use goto into other onstructions[59]

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

does not protect directly versus luckypatcher but in case of an custom implementation it makes the process of analyzing the app more time consuming in theory a good addition to the security of the application, but against luckypatcher directly since it works on java level in order to disguise the way the code works it enforces increased initial effort an attacker has to spend in order to understand the code and thus reduces the likelihood of attackers to being motivated to crack the application

Proguard

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

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

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

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

after proguard is finished dx converts to *classes.dex*

[63]

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
[72]

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 [45] [86]

Dexguard

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

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

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 [86]

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

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

Allatori

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

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

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

[2] [3]

4.1.5 Content Server

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

The implementation can be described with the application Spotify [76] as a reference. 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, the music in this case, is no longer on the phone itself, but streamed from the server. The attacker still can circumvent the user verification inside the application, but they are facing a second layer of security afterwards. Since the content is on the server and the user has to be authorized on it, no content is available

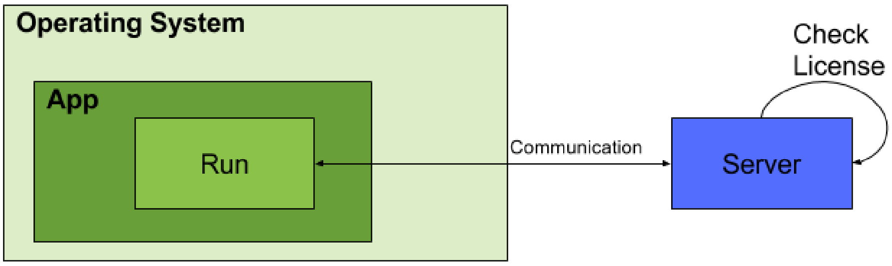


Figure 4.2: Abstraction of an application and a content server

inside the application. Thus attacks on the applications itself do not work anymore that easily.
Having a content server is good solution against piracy but it has downsides as well. The first problem is that this model cannot be applied to all applications universally. It means that it must be possible to extract parts of the application's logic and implement them on a server. This is not possible for all applications. The second problem are the additional resources needed. When outsourcing parts of the application on a server, not only knowledge and money is needed for the server, but an additional application for the server has to be created as well. Not every developer can handle this additional workload. The third problem is the resulting always online necessaty which limits the freedom of users and creates additional unnecessary traffic. This is not accepted by all users.
Nevertheless, if this implementation can be realized, it is safe from Lucky Patcher autopatching as well as custom patches. In addition, this mechanism protects the developers IP when the core algorithm is moved to the server. This prevents attackers not from only using the application for free, but also from reconstructing the the core functionality and implementing it somewhere else, thus the application gives less incentives for attackers.

4.2 Encryption

A part of the license verification's insecurity is the fact that not only the possible outcome is binary, either the license is verified or it is not, but it is also possible to predict the outcome since in the end, only one result is accepted. This mechanism can be easily attacked. The attacker can modify that even the wrong result is always accepted.
In order to fight this major flaw, encryption is introduced to extend the original license verification. The advantage of encryption is the unpredictable outcome of an input

which is not spoofable anymore compared to the binary check. Before taking advantage of encryption, it has to be decided what content should be encrypted and where the key should be stored.

Encryption

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

Resource Decryption

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

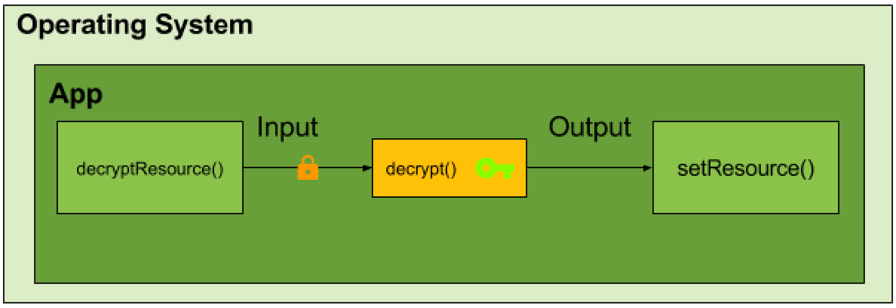


Figure 4.3: Encrypted resources which have to be decrypted on startup

Action Obfuscator

The second approach is to use encryption as obfuscation. The idea is to have a method, which is called by each method, to 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. In order to make it harder to circumvent the encryption of the methods, objects can be encrypted as well. An abstract presentation of the mechanism can be seen in figure 4.4.

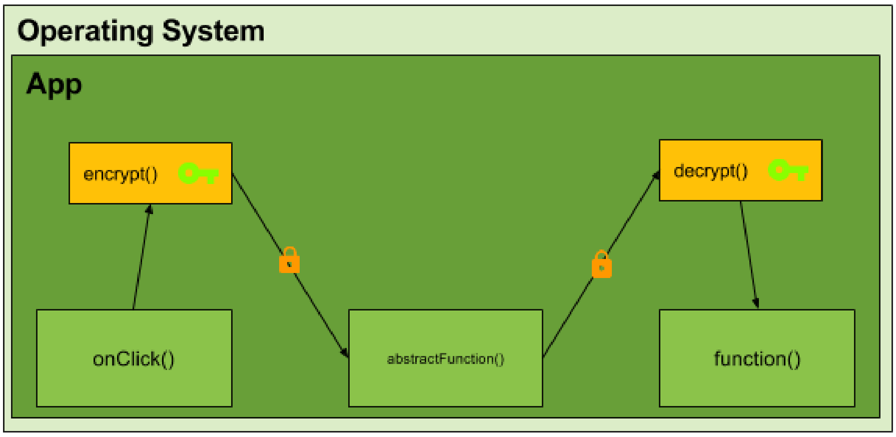


Figure 4.4: Encrypted actions to obfuscate dependencies

Communication Decryption

The third approach is to use encryption on the server response as seen in figure 4.5. This is an additional security feature which is applied in combination with a content server described in subsection 4.1.5. When the user does the login on the server, additional unique device specific paramters have be passed as well. On the first login, the server generates a cryptographic key which is used for communication with the user on this specific device. The corresponding key can either be generated on the device or be shared by the server. This mechanism allows only authorized users on a specific device to decrypt the communication.

A similar approach is used for streaming DRM protected content on Android. The encrypted content can only be decrypted by a native interface provided by the OS which stores the decryption key. [13]
This methodology focuses on the security of the content instead of the application itself.

Encryption

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

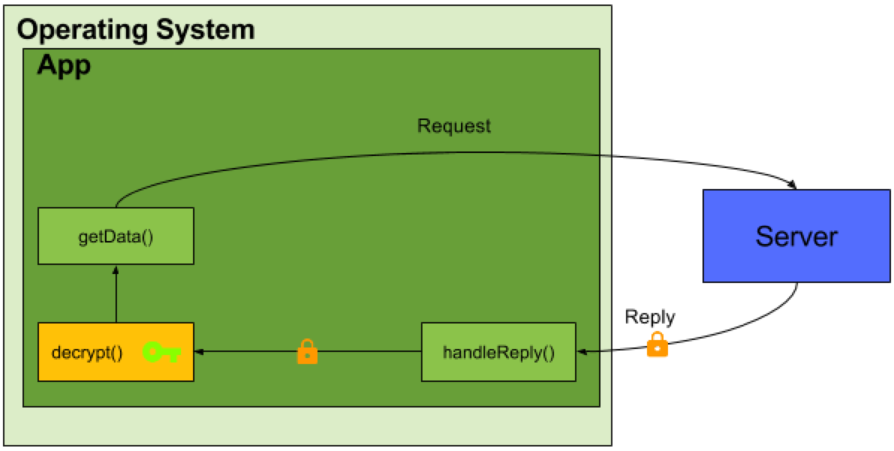


Figure 4.5: Encrypted communication with a server

Secure Element

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

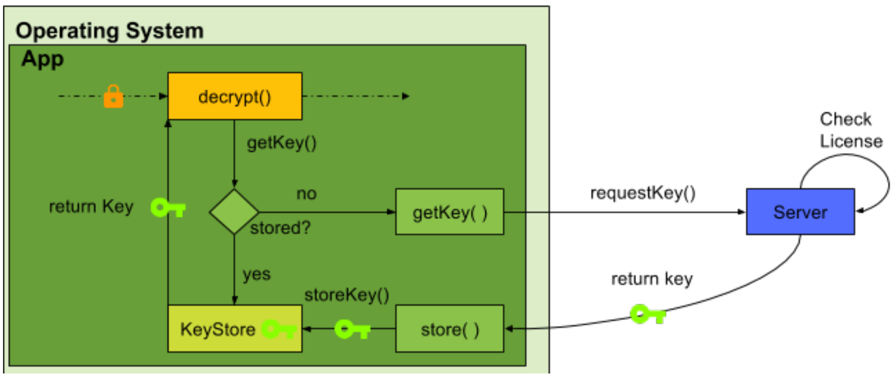


Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

Secure Element

Since there is the possibility to read the cached encryption key [84] and crack the encryption, the use of Secure Element (SE) is proposed. A secure element is a tamper-resistant platform which can be used to securely host applications and cryptographic keys [47]. There are different form factors for SEs. For Android, the microSD form factor is the interesting one. It can be either mounted in the microSD card slot or by using an adapter on the USB interface, which requires the device to support USB On-The-Go (OTG) [**usbOtg**]. The resource is accessed over reads and writes to the filesystem. Since the SE has to be small to fit the size of an microSD card and powered by the host system, its hardware capabilities are constrained. The result is a performance of 25MHz which does not allow complexe comnputations. [77]
For this reason the usage of the SE is restricted to simple tasks, like storing a key used for decryption. The advantage of an SE is that its functionality is outside of the Android application and thus cannot be manipulated by Lucky Patcher.
An abstract presentation of the use of a SE can be seen in figure 4.7.
Integrating a SE comes with some problems.

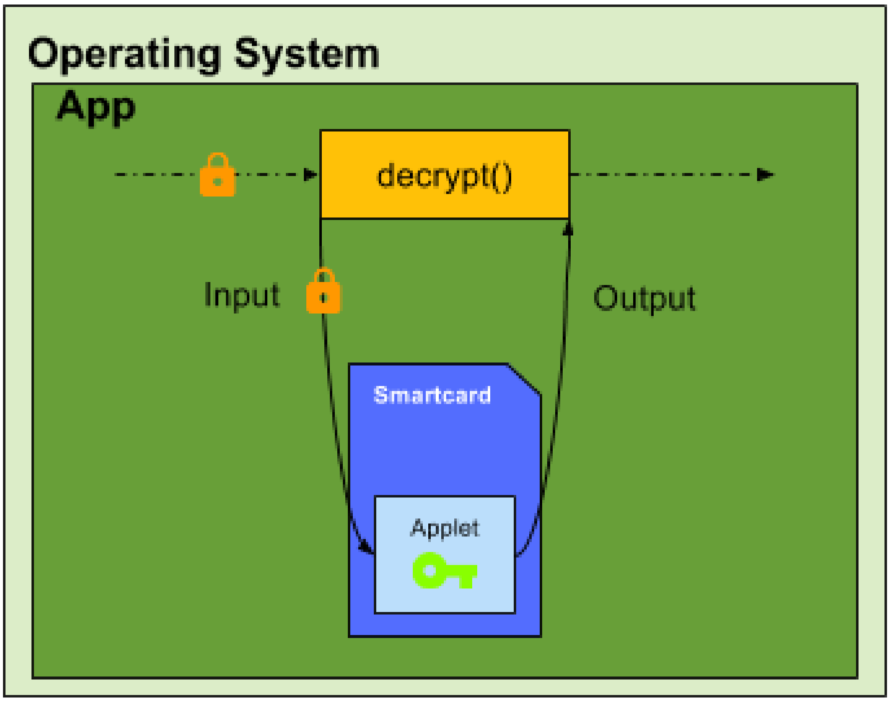


Figure 4.7: Decryption by using a smartcard

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

The first problem is that the user is required to buy extra hardware. This means the user has to spend extra cache and to have the SE always around. The connection to the device using a cable is not the most convenient solution as well. The second problem is that not all devices either have an microSD card slot, nor OTG support. For example, the Nexus 7 (2012) and the Nexus 6P neither have the capability to use a microSD card. While the Nexus7 was supposed to have OTG, it did not work with the used SE, while the Nexus 6P did not support OTG out of the box. Both devices needed even needed additional plugins to read the OTG mounted microSD in a file explorer. The third problem is that each manufacturer implements its own interpretation for the interface which makes SE incompatible to each other. For this reason, the SD Association proposed the smartSD in order to have a universal standard for SEs [73].

se signiert mit key+android_id welche unique ist

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

-

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

4.3 Extend Android Runtime

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

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

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

already on the way, cannot be done from one day on the other, but right now not a protection against luckypatcher, will only be a solution when art code included in apks but why not now? Evaluation Why is Android not all ART now? Your applications still compile into Dalvik (DEX) code, Final compilation to ART occurs on the device, during install, Even ART binaries have Dalvik embedded in them, Some methods may be left as DEX, to be interpreted, Dalvik is much easier to debug than ART
[59]

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

art is a far more advanced runtime architecture, brings android closer to ios native level performance vestiges of dex still remain to haunt performance, dex code is still 32 bit very much still a shifting landscape, internal structures keep on changing, google isnt afraid to break compatibility, llvm integration likely to only 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
[59]

5 Conclusion

This thesis analysis the Lucky Patcher and its attacks on the different license verification libraries. Android's open architecture allows the user to extract and install applications from any sources. The freedom comes at the price that Lucky Patcher can modify APKs even without root permission. This puts the unchanged implementation of license verification libraries at risk of being voided.
Google is aware of the situation but cannot do more than to motivate the developers to make their library implementation unique.

5.1 Summary

The scope of this thesis was to analyse how Lucky Patcher is carrying out the attack on the license verification libraries and what countermeasures developers can apply to protect their application against it.

The first chapter starts with the introducion of software licensing, its goals and the reason it is enforced. The current situation and problems with licensing on Android is portrayed. Different approaches to improve and enforce license verification are presented in the related work.
The second chapter explains the fundamentals needed to understand why software piracy is a problem. Android and the steps needed to run an application are explained. This chapter introduces the license verification libraries which are target of Lucky Patcher and tools used for the analysis.
The third patcher is all about the Android cracking tool Lucky Patcher. First the functionality is presented. Then an analysis of the application itself is done followed by a blackbox analysis and the evaluation of the result. In the end the lessons learned from the analysis are pointed out.
The fourth chapter suggests three different types of countermeasures. The first part is about improvements to the current state of the license verification libraries and the addition of integrity checks. The second part introduces outsourcing of content and encryption as a non predictable implementation of license enforcement. The third part suggests improvements in the environment to protect against cracking tools.

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 [63] 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 [72] not a question of if but of when bytecode tool to generate the licens elibrary on the fly, using random permutations and injecting it everywhere into the bytecode with an open platform we have to accept a crack will happen [55]

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

alles hilft gegen lucky patcher auf den ersten blick, jedoch custom patches, welche Lucky Patcher anbietet[63], 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

close down free installations

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 [63]

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] Androguard. *Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)* URL: https://github.com/androguard/androguard (visited on 03/03/2016).

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

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

[12] Android. *Dalvik bytecode*. URL: https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html (visited on 02/29/2016).

[13] Android. *DRM*. URL: https://source.android.com/devices/drm.html (visited on 02/29/2016).

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

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

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

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

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

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

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

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

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

[23] Android Developers. *Signing Your Applications*. URL: http://developer.android.com/tools/publishing/app-signing.html (visited on 03/01/2016).

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

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

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

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

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

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

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

[31] 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).

[32] 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).

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

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

[35] 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).

[36] 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).

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

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

[39] CVE. *Common Vulnerabilities and Exposures*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=android,+privileges (visited on 02/25/2016).

[40] CVE Details. *Google Android: List of Security Vulnerabilities (Gain Privilege)*. URL: http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/opgpriv-1/Google-Android.html (visited on 02/29/2016).

[41] 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).

[42] Egor F. *64-bit Android and Android Run Time*. URL: https://software.intel.com/en-us/android/articles/64-bit-android-and-android-run-time (visited on 03/01/2016).

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

[44] Erikrespo. *Häufigkeit der verschiedenen Android-Versionen. Alle Versionen älter als 4.0 sind fast verschwunden*. URL: https://de.wikipedia.org/wiki/Android_(Betriebssystem)#/media/File:Android_historical_version_distribution_-_vector.svg (visited on 02/25/2016).

[45] 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).

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

[47] GlobalPlatform. *GlobalPlatform made simple guide: Secure Element*. URL: https://www.globalplatform.org/mediaguideSE.asp (visited on 02/29/2016).

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

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

[50] B. Gruver. *smali/baksmali*. URL: https://github.com/JesusFreke/smali (visited on 03/03/2016).

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

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

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

[54] E. Johnston. *Mobile Game Piracy Isn't All Bad, Says Monument Valley Producer (Q&A)*. URL: http://recode.net/2015/01/06/mobile-game-piracy-isnt-all-bad-says-monument-valley-producer-qa/ (visited on 01/18/2016).

[55] Kevin. *How the Android License Verification Library is Lulling You into a False Sense of Security*. URL: http://www.digipom.com/how-the-android-license-verification-library-is-lulling-you-into-a-false-sense-of-security/ (visited on 01/18/2016).

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

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

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

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

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

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

[62] 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).

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

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

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

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

[67] B. Pan. *dex2jar - Tools to work with android .dex and java .class files*. URL: https://github.com/pxb1988/dex2jar (visited on 03/03/2016).

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

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

[70] Runtastic. *Runtastic PRO Laufen & Fitness*. URL: https://play.google.com/store/apps/details?id=com.runtastic.android.pro2&hl=de (visited on 01/20/2016).

[71] Samsung. *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).

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

[73] SD Association. *smartSD Memory Cards*. URL: https://www.sdcard.org/developers/overview/ASSD/smartsd/ (visited on 02/29/2016).

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

[75] skylot. *Dex to Java decompiler*. URL: https://github.com/skylot/jadx (visited on 03/03/2016).

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

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

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

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

[80] statista. *Number of apps available in leading app stores as of July 2015*. URL: https://its.uncg.edu/Software/Licensing/ (visited on 01/16/2016).

[81] P. Steinlechner. *Cracker beißen sich die Zähne an "Just Cause 3" aus*. URL: http://www.sueddeutsche.de/digital/illegale-kopien-von-computerspielen-cracker-beissen-sich-die-zaehne-an-just-cause-aus-1.2810482 (visited on 01/26/2016).

[82] Stericson. *Busybox - Android-Apps auf Google Play*. URL: http://www.androidheadlines.com/2010/10/amazon-send-developers-a-welcome-package.html (visited on 01/19/2016).

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

[84] P. Teoh. *How to dump memory of any running processes in Android (rooted)*. URL: https://tthtlc.wordpress.com/2011/12/10/how-to-dump-memory-of-any-running-processes-in-android-2/ (visited on 02/29/2016).

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

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

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

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

[89] 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).