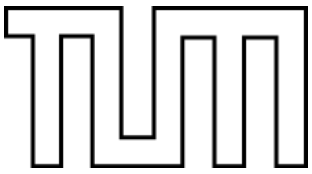


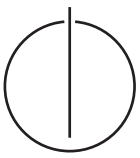
DEPARTMENT OF INFORMATICS

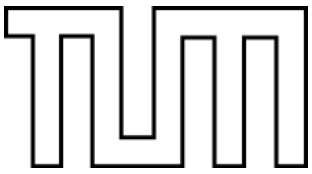
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Johannes Neutze, B. Sc.





DEPARTMENT OF INFORMATICS

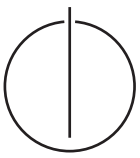
TECHNISCHE UNIVERSITÄT MÜNCHEN

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
- some patterns can be tricked with mopdification but reverse enineerint to easy
- freedom vs walled garden, only able to stop piracy when certificate is checked
- 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 [90].
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 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 agreements are implemented. This includes Digital Right Management solutions which deny access to the software in case of a wrong serial key or unregistered account.

The problem is that these mechanisms do not offer absolute security and pirates always try to circumvent them. This results in an everlasting race of arms between software creators and software thieves [91].

1.2 Motivation

Licensing is also present in Android. With a market share of almost 82.8% in Q2 of 2015 [58] 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 [34]. This giant number of Android devices is powered by Google Play [55]. 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 [94]. 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 [65].
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 [76]. 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 [96] and express their need to protect their IP on platforms like xda-developers [83] or stackoverflow [88]. Many of the developers have problems with the license verification mechanism and name *Lucky Patcher* as one of their biggest problems [89].

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.
In his master's thesis "A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks" [30] Bernhard takes a look at license verification an in-app billing 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 for further investigation on this topic.
Muntean's master's thesis "Improving License Verification in Android" [71] 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.
Other scientific papers, like Jang, Ji, Hong, et al. in their paper "Protecting Android Applications with Steganography-based Software Watermarking,"

2 Foundation

Before understanding the attack mechanisms and discussing countermeasures, necessary background knowledge has to be provided. Consequences 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 [28]. Software piracy is defined as the 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 Problems for 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 and no revenue is generated. It is even worse when the pirated application is sold in another app store. In this case the pirate gets the profit which should be the developer's.
Revenue is not only lost when the application can be downloaded for free. There is also follow up revenue effected, when an applications is changed. There are two main types of indirect revenue. /newline The first type is in-app purchases. They are a popular source of income for so called freemium 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 with in-app purchases like cosmetic interface changes 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 in-app purchase thus no earnings are generated for the

developer while the user can access the content.
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 [53]. 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

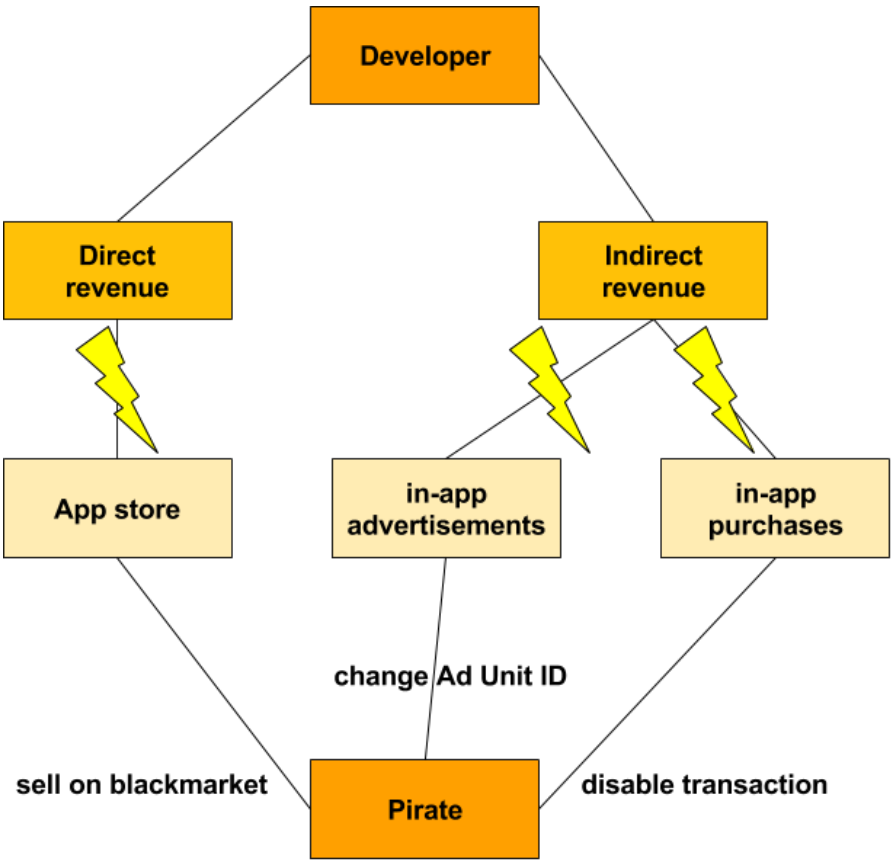


Figure 2.1: Different ways to generate revenue and how the pirate can cut them

black market store or website and distributed without the environment of an official app store. This 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 bugs. Users which do not know they are using a pirated version will connect the unsatisfying behaviour to the developer. This results in the

loss of reputation and revenues not even connected to this application.
If an application uses server resources, the developer can neither monitor the growth in the usage of their application nor do they get the money to upgrade it [68].
Developers make a living of their applications. When they do not make a profit, 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 Dangers for Users

The loss of developers in the ecosystem is bad for users, but they can also be harmed by piracy. Users prefer pirated applications, they seem to be free of charge, but that's not necessarily true. 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 and missing updates when related from an unofficial source. In general, the risk is very high that pirated software offers a user experience worse than the original. [33] [68]
Pirated software is always a risk and should not be installed since the integrity of the application cannot be ensured without deep inspection.

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 [45]. 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, like Blackmart [31], and websites offering cracked Android Application Package (APK), such as crackApk [41]. 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 cheated. Software pirates practice professional theft and expose users to risks (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% on a given day. [83] [96] 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 [61].

Android applications are at an especially high risk for piracy because of their use of bytecode, which is an easy target to reverse engineering as shown in the further proceeding.

2.2 Android

Android is an open source 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, but also runs on MIPS and x86 processors. The following gives an overview over the architecture of Android and a deeper insight in the runtime system of Android. The architecture of the Android software stack can be seen in figure 2.2.

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

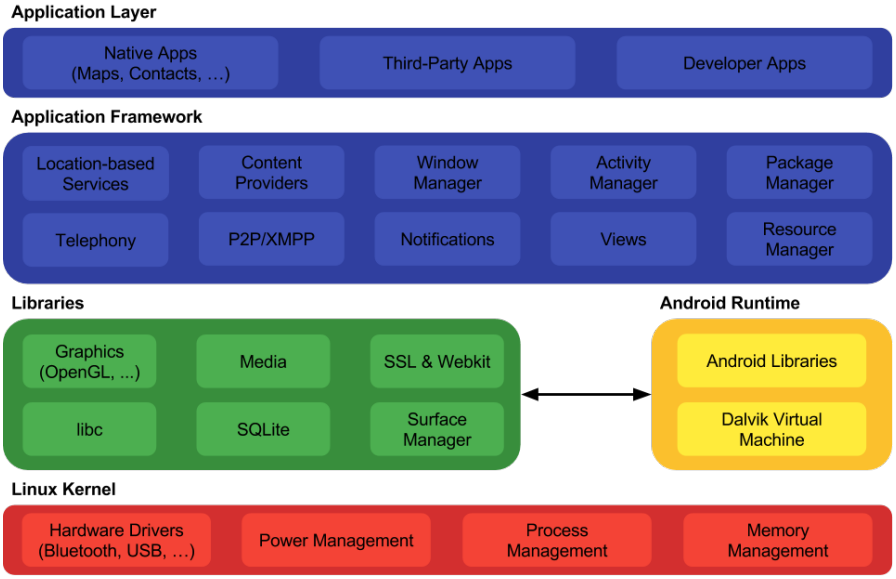


Figure 2.2: Android's architecture [72]

management 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.

On top of the libraries and the runtime lies the application framework. This layer provides generic functionality to applications over Android's Application Programming Interface (API), such as notification support.
The top layer is responsible for the installation and execution of applications.
Using these abstraction layers allows 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 4.1.3. When all Java classes are compiled to .class files, they are packed into a Java Archive (.jar) file.
Since Android is using a different Virtual Machine (VM) for executing the code, the Java bytecode has to be converted to Dalvik bytecode. The Android Software Development Kit (SDK) provides *dx*, the tool used to convert .class files to a single *classes.dex* file. The VM and the Dalvik EXecutable (.dex) file format will be described in the following. Similar to the Java bytecode, obfuscation can be applied.
The APK itself consists of three parts.

- *classes.dex*, containing the bytecode
- 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 is done similar to the Java code signing [48]. It ensures the integrity and authenticity of the APK. Tampering

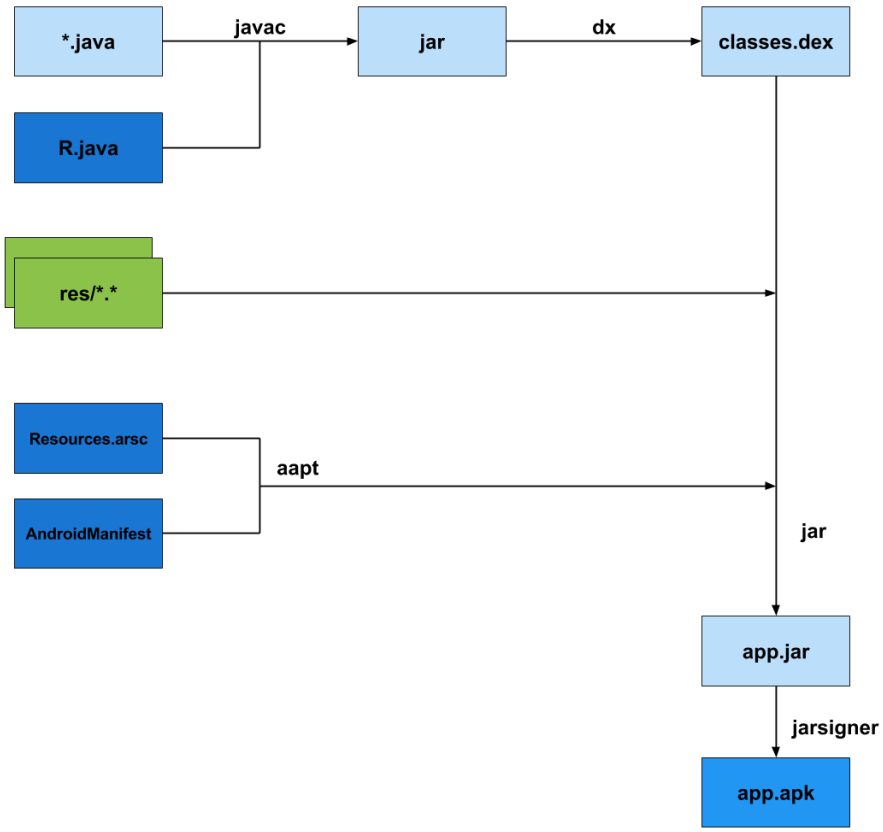


Figure 2.3: APK build process [67]

can be detected and updates for applications with the same name and certificate can be installed. Afterwards *zipalign* is used to mark uncompressed data. [23] [67]

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 configuration data, e.g. the signature [73]. While the static resources, like drawables and layouts, are in the res folder, the resources.arsc contains the compiled resources. Natice libraries is stored as **.so* files in the libs folder, split by the different processor types, like armeabi-v7a for ARM or x86 for Intel processors. 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). [63] [47]

```
|-- 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

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. [11] In case there are 64 bit values, adjacent registers are used to store it.
Similar to Java bytecode, instructions are not stored inside the method but as a reference pointing to the variable. While in Java each class has its constants, like numbers, strings and identifier names, grouped together in heterogenous pool (see figure figure 2.5, left side), Dalvik bytecode uses one pool for each type. When compiling Java bytecode to Dalvik bytecode, the heterogenous pools of each Java class are merged together in one global pool for each type (see figure figure 2.5, right side). In this process, duplicates in a pool are removed, which reduces memory need for constant but increases the number of references. This is most effective for strings. A decrease of the memory footprint of up to 44% compared to the .jar is possible.

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

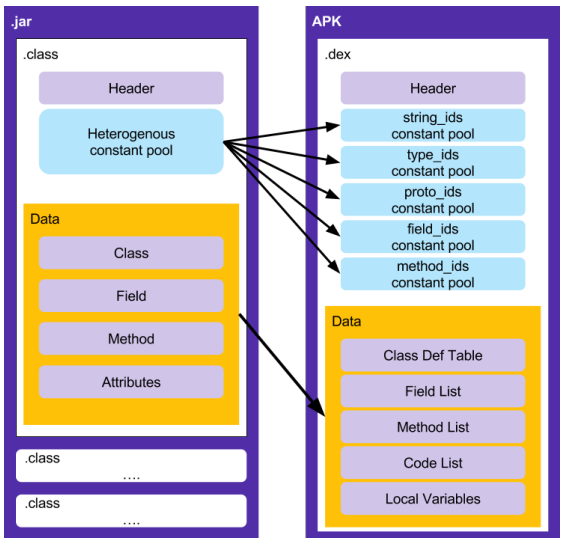


Figure 2.5: .jar to APK transformation [32]

have to be updated. [17] [47]

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

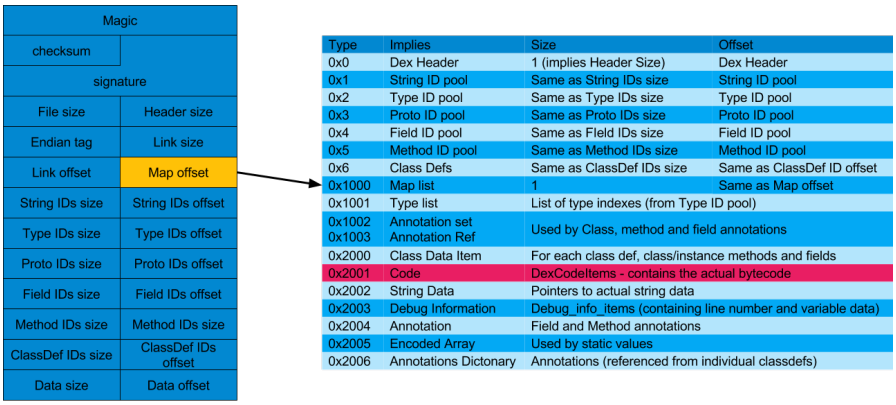


Figure 2.6: .dex file format [67]

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, the APK, containing the code, 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 its integrity and authenticity. The *META-INF* folder contains the necessary files for performing the signature verification process. The three files are the *MANIFEST.MF*, the *MANIFEST.SF* und the *CERT.RSA*. The manifest, *MANIFEST.MF*, contains the name and the digest of each file inside the APK, e.g. *classes.dex* and its signature. The the signature file, *MANIFEST.MF* which is used for the code signing. It is similar to the manifest and contains the SHA1 digest of the manifest and the digests of each entry inside the manifest. The *CERT.RSA* is the digital signature, called signature block, which is used to sign the files. While the manifest files are used to detect tampering while the digital signature is used to identify the code signer. In order to apply updates, the old and the new APK need to have the same package name and code signer. Since Android allows self-signed certificates, no connection of whether the APK was signed by the right person nor whether the application is allowed to be installed can be closed from the digital signature [48] [98]
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. [63]
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 separate user ID. This way each application runs separated from the others and has no access on resources except its own. [15]

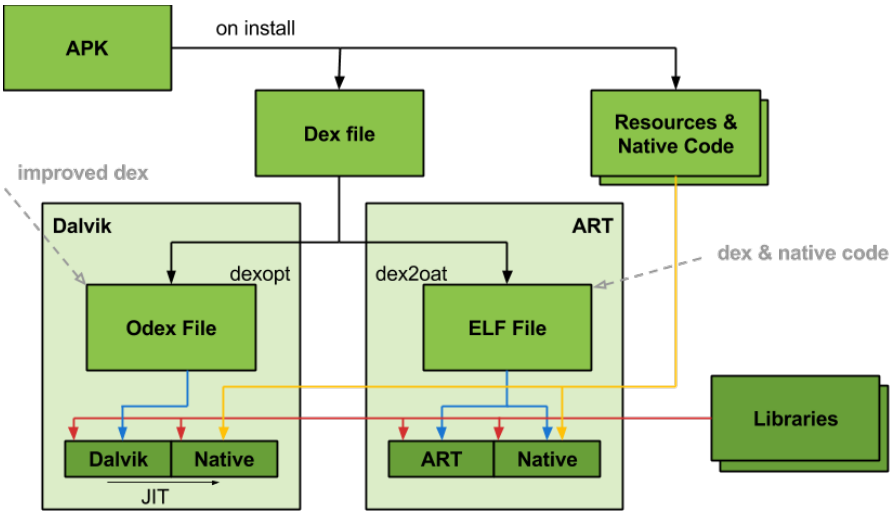


Figure 2.7: Installing an APK on a device [26]

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 Icelandic town and introduced along with Android in 2008 [24].
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 approximately 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 separated from objects and optimizes application startup by using zygotes. [47] [67]
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 compatibility, ART still works with bytecode in the .dex files format [9]. With the release of version 5.0 *Lollipop* ART it became the runtime of choice since DVM had some major flaws. Throughout the Android 6.0 *Marshmallow* previews it was constantly evolving and sometimes breaking with older versions at the cost of almost no documentation.
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 portability. In practice the preference is to compile to native since adding LLVM bitcode adds another layer of complexity to ART.
Different from DVM is the fact that ART uses not one but two file formats. Similar to the zygote of DVM, ART offers an image of pre-initialized classes and and related object at run time, the boot.art file. It is poorly documented and still changing 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. [67] [10] [9] [46]
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 February 2015. Details and references of OS vulnerabilities can be accessed on pages like Common Vulnerabilities and Exposures or similar [42] [43].
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 [99]. 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*. The phone is then nonfunctional since the software cannot be executed anymore. [70]
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.

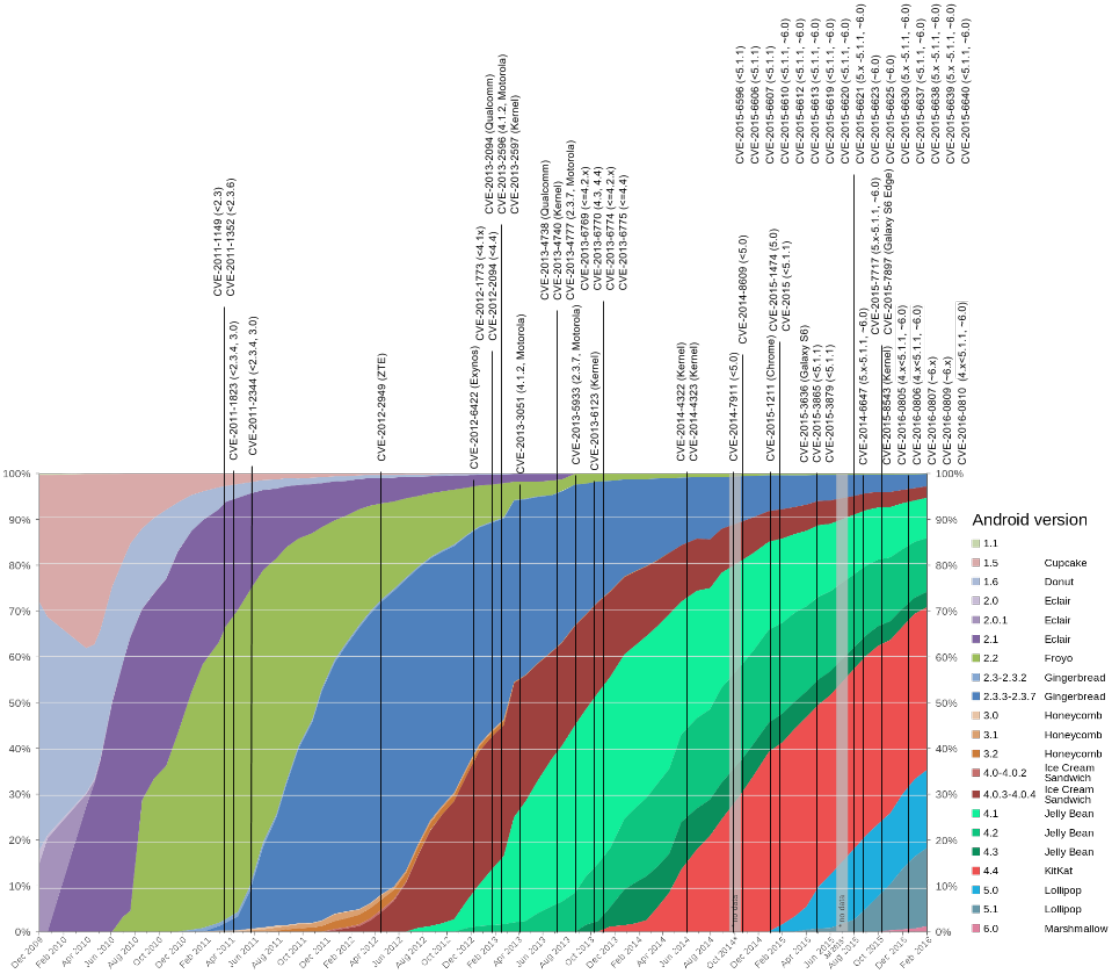


Figure 2.8: Timeline of market share of evolving Android versions and dates of root exploits recorded [49] [1] [42] [43]

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 and niche stores as well.
The scope of this thesis are Google, Amazon and Samsung since they have a critical mass in users and a copy protection mechanism. This is the reason, Lucky Patcher focuses them.
All these stores have to fight the copy protection problem in order make their store attractive and attract developers by having 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 License Verification Library (LVL) on the 07/27/2010 [38]. It is easy to use and free of charge. The documentation can be found on the Android developers website [19].
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

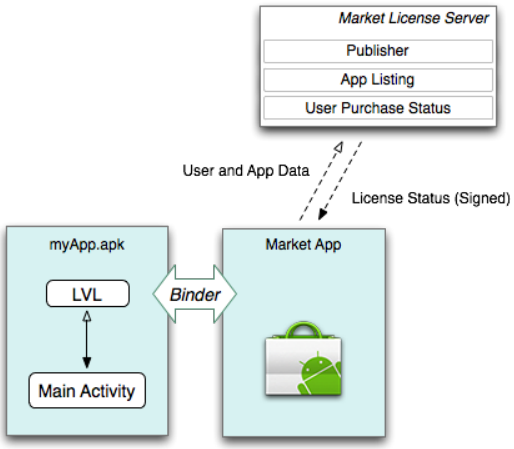


Figure 2.9: Google's implementation of license checking [19]

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. [62] [19]
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] [13]
The second extension is the callback for the asynchronous handling of the license

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

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

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. [19] [22] [13] [62]
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

```
133     private class MyLicenseCheckerCallback implements LicenseCheckerCallback {
134
135         @Override
136         public void allow(final int reason) {
137             ...
138         }
139
140         @Override
141         public void dontAllow(final int reason) {
142             ...
143         }
144
145         @Override
146         public void applicationError(final int errorCode) {
147             ...
148         }
149     }
```

Code Snippet 2.2: LVL license check callback

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 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. [19] [22] [13] [62]
Upon execution, the information is passed to the Google Play Service client. The

```
57          final String mAndroidId = Settings.Secure.getString(this.getContentResolverSettings.Secure.
                ANDROID_ID);
58          final AESObfuscator mObsfuscator = new AESObfuscator(SALT, getPackageName(),
                mAndroidId);
59          final ServerManagedPolicy serverPolicy = new ServerManagedPolicy(this, mObsfuscator);
60          mLicenseCheckerCallback = new MyLicenseCheckerCallback();
61          mChecker = new LicenseChecker(this, serverPolicy, BASE64_PUBLIC_KEY);
62
63          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. [19] [22] [13] [62]
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. 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 the resulting policy, internet connection is needed. [19] [22] [13] [62]

2.3.2 Amazon DRM (Kiwi)

Amazon started its own application store in October 2010 [6] as an alternative go Google Play. The Amazon appstore opened to the public on the 03/22/2011 [7]. 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

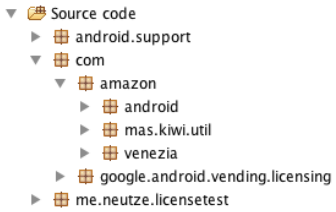


Figure 2.10: Amazon library structure in decompiled application

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." [5].
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. [5]
The implementation can be analysed with reverse engineering. The license verification

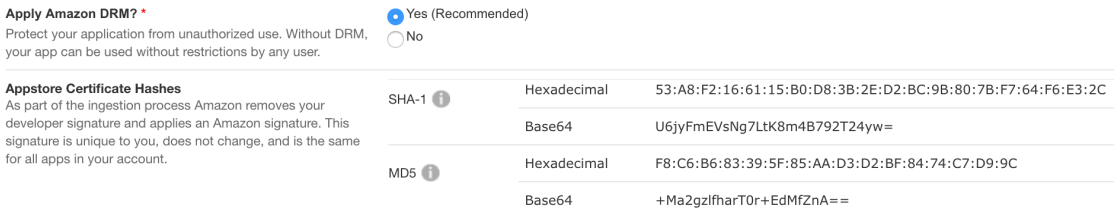


Figure 2.11: Developer preferences in the Amazon developer console [5]

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

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

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

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 [39]. 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* [80].
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 [80]. 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 implementation of the *LicenseCheckListener*. It contains the

```
12  ...
13  <uses-permission android:name="android.permission.INTERNET" />
14  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
15  ...
```

Code Snippet 2.5: Include permission in theAndroidManifest.xml [80]

two 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.

```
85          @Override
86          public void licenseCheckedAsValid() {
87              mHandler.post(new Runnable() {
88                  public void run() {
89                      ...
90                  }
91              });
92          }
93
94          @Override
95          public void licenseCheckedAsInvalid() {
96              mHandler.post(new Runnable() {
97                  public void run() {
98                      ...
99                  }
100             });
101         }
```

Code Snippet 2.6: Zirconia license check callback

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

Code Snippet 2.7: Setting up the Zirconia license check call

2.3.4 Abstraction

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. In the further analysis it will be shown that this binary mechanism is an easy target for attacks, such as executed by Lucky Patcher.
The functionality of the libraries can be abstracted as a simple *yes/no* check as seen in figure 2.12.

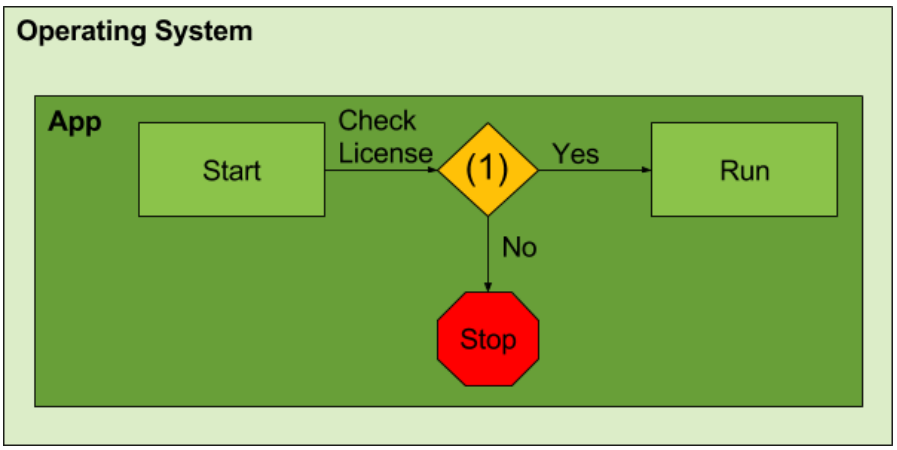


Figure 2.12: Abstraction of the current license verification mechanism. The library is represented by (1)

2.4 Code Analysis

Cracking tool are modifying the application code since it contains the license verification. In order to understand the attack, the APK's *classes.dex* is analysed. The goal is to identify how the exact mechanism applied to circumvent the license verification mechanism. The acquired knowledge is later used to suggest countermeasures against cracking tools.
The code is reverse engineered on different abstraction layers to not only identify the local bytecode change but also to retrace the change in the functionality of the code. While dex and smali code reflects the local changes, Java code is used to see the bigger picture. The tools to retrieve the different abstractions and the way the changes are discovered will be described in the following.

2.4.1 Retrieving an APK

The analysis is performed using different tools on a desktop computer. The APK has to be extracted from the phone and transferred onto the computer because the tools cannot analyse the application while it is still on the phone. This is done using the *adb shell* which is part of the Android SDK.
The *adb shell* can be used inside the computer's command line tool. The Android device must be connect to the computer via USB and USB debugging has to be activated in the device's developer settings. The *adb shell* is used to access the filesystem of the device in order to pull the target APK onto the computer. The user has no read rights

on the folder and thus cannot see or list the content of the folder. In order to pull the desired application package, location of the application must be known to the user. The package manager can be used acquire location of the application by listing all installed applications and their location. The list is retrieved using *adb shell 'pm list packages -f'* in the command line tool. The result looks contains one application per line, e.g. *package:/data/app/com.ebay.mobile-1/base.apk=com.ebay.mobile*. The information is used in the *adb shell* to pull the application to the computer by executing *adb pull /data/app/com.ebay.mobile-1/base.apk* in the command line tool.
In case the user has *root*, it is possible to change the permissions of the folder. This way the folder is visible and accessible in a suited file explorer application and can be send to the computer.

2.4.2 Code Abstraction Levels

The code of the application will be inspected on three different abstraction levels.
The first level is the original dex bytecode contained in the *classes.dex*. It is target of the attack and is used to detect the changes done by Lucky Patcher on bytecode level.
The second level is the smali code. It is used to represent the dex bytecode in a readable way and to identify the the result of the changes.
The third level is the Java code. It gives the best overview of the code. The result of the changes in the context of then method or class can be best identified in this presentation.
The Java code can be decompiled from the dex bytecode since the build process can be reversed as seen in figure 2.13.

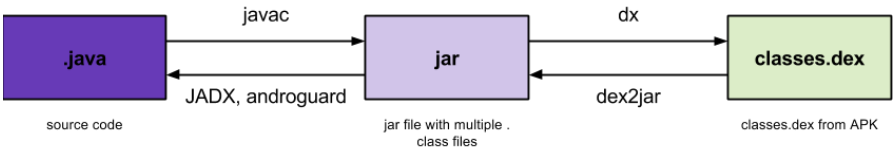


Figure 2.13: Java .class and .dex can be transformed bidirectional [67]

dex Analysis

The *classes.dex* contains the application code and has to be modified by the cracking tool to carry out the attack. Analysing the dex bytecode allows to point out which characters in the bytecode have been modified.

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 destination* as seen in line 3. The extracted *classes.dex* is still contains the code in binary. Hexdump is used to convert the code into a hexadecimal view.
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. 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 better identify opcodes and translate them using an opcode table [74]. For example, 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.*.

```
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

Smali Analysis

The smali code is disassembled from dex bytecode using *baksmali* [56]. This is done in order to interpret the changes in the bytecode regarding functionality.
Assembling and disassembling of dex and smali is possible without the loss information since they have a bijective mapping[56]. The syntax is loosely based on Jasmin's syntax It takes the APK and disassembles its *classes.tex* file. The output is a file smali file for each class. Smali has two advantages over dex bytecode. The first advantage is the replacing of opcodes with their actual opcode name. The second advantage is the reconstruction of the class and method structure. This makes it easier to analyse the

code. This process is done using the script in code snippet 2.10.
An example of smali code can be seen in code snippet **??**. It is easier to understand than

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

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

the dex presentation. The content of variables, as in line 3, can be identified without big effort. 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

Java Analysis

The goal of the Java code is to reverse engineer the original code including the changes of the attacl. It suits best for the analysis of the new behavior 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.
DAD, short name of "DAD is A Decompiler". It is part of Androguard [8]. It works with the dex bytecode and does not require third aprty tools like dex2jar [75]. Code

snippet 2.12 shows how it is used to decompile the APK into Java code.
JADX [84] is the second compiler. THe decompilation is directly done from the APK's

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

Code Snippet 2.12: Script to decompile to Java using androguard

dex bytecode to Java code. It is the command line as seen in Code snippet 2.13.

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

Code Snippet 2.13: Script to decompile to Java using JADX

2.4.3 Comparison of Code using Diff

The amount of code generated for the abstraction levels cannot be analysed without extra tools. 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 in order to receive the differences between them.
The changes Lucky Patcher applied have to be identified. This is achieved by comparing the different code abstractions of the original APK with the cracked application. Diff is used in a script to generate the result for different applications and abstraction levels at once (see code snippet 2.14). Doing this automatically and using diff saves a lot of time. The result does not only contain the change as original and new code, but also the location where the change happened. 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

```
1 #!/bin/bash
2 #dex
3 diff -r /dex/original/ /dex/manipulated/ > dex.diff
4 #smali
5 diff -r /smali/original/ /smali/manipulated/ > smali.diff
6 #dad
7 diff -r /java/dad/original/ /java/dad/jadx/ > dad.diff
8 #jadx
9 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

3 Cracking Android Applications with Lucky Patcher

The cracking of applications on Android is a widespread phenomenon these days since the modification of the dex bytecode is possible. There are a number of tools attacking and altering the license verification library of premium applications. Many developers discuss this piracy threat on the internet and the tools used.
This thesis will focus one of the most popular cracking application, Lucky Patcher, and its license verification bypassing attack.

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" [36]. It is written by a developer called ChelpuS and currently on version is 6.0.7 (03/03/2016).
Lucky Patcher offers different patches which can be used to modify an application. They can be applied to remove the license verification in premium apps or Google Ads and to change or restrict permissions and activities [36].
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. [71]
Using Lucky Patcher is fairly simple. The application can be downloaded as an APK from the official website [36] and installed on the device. After the launch of Lucky Patcher, all installed applications are shown in a list. A colored text indicates what patches are available and can be applied to an application. In order to have unlock all features, the device has to be rooted.
When an application is selected, a submenu offers various actions, e.g. to get information about the app or run it. The patches menu opens the submenu of figure 3.1, on the left. It shows the available patches for this application.
There are two types of patches. The first type is the *custom patch* and the second type are universal patches. Their goal is the disabling of the license verification libraries, removing Google Ads and rebuilding of the application to use the emulated LVL and

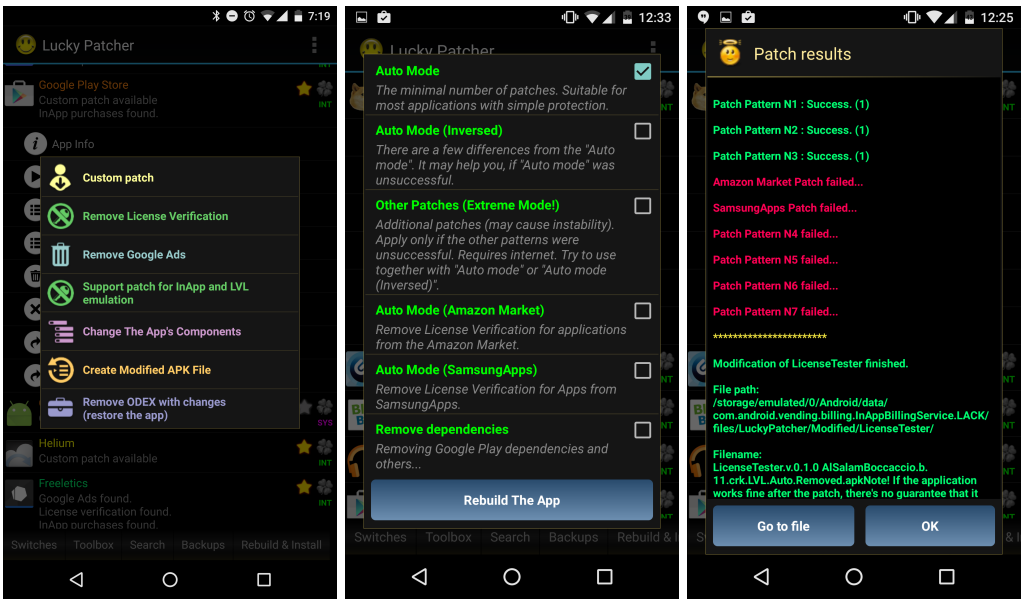


Figure 3.1: Left to right: Features offered LuckyPatcher, modes to crack license verification and the result after patching

in-app billing APK.
Lucky Patcher offers two different approaches to apply these patches. The first approach is to apply them directly on the device and requires *root*. This method creates an .odex version of the patched application in the Dalvik cache on the device. The second approach is the creation of a modified APK. The same patches as in the first approach are available after selection but they are not applied directly on the phone. This approach extracts the application of choice from the storage, applies the selected modifications and creates a new APK. This cracked application can either be distributed to other devices or installed on the device after removing the original APK.
The custom patch is the most powerful modification. It applies changes specifically designed for the chosen application. The custom patches have to be provided by users who reverse engineered the application and created a solution which can be applied with Lucky Patcher. The changes are either applied by replacing the original **.so* file of the native library with a cracked one or by injecting a bytecode sequence into the application, disabling the desired feature.

This thesis focuses on the *Auto Modes* Lucky Patcher provides for patching license verification libraries. The goal of circumventing the license check is to make the pirated application work as if it had been legally acquired from a corresponding store.

There are six automatic modes available as seen in figure 3.1 in the middle. Their description is rather short and does not offer information of how the modes are applied and working.
When a mode is chosen the patching starts. The result is shown when the process is finished. As seen in figure 3.1 on the right, it is indicated that different patching patterns are used to remove the license verification. The different patching patterns are analysed and explained in section 3.4. /newline Lucky Patcher does guarantee that the result is working and all license verification related restrictions are removed.
The way this is achieved is analysed in two ways.

- inspection of the source code
- blackbox analysis using cracked applications

3.2 Code Analysis

The inspection of Lucky Patcher's source code of version 6.0.4 was performed using the two decompilers DAD and JADX, which are described in subsection 2.4.2. The reverse engineered code was analysed using a text editor, like Atom [51].
First, the folder structure was inspected by loading the reverse engineered code into the editor. On the first look, different folders can be spotted in the applications tructure. These folders contain the resources, different libraries and the actual Lucky Patcher code. The folders containing code can be divided into four categories.

1. Android Support Library v4 with many of it's modules - The library is used to provide downward compatibility of Android related functions.
2. Lucky Patcher - Code containing the Lucky Patcher logic are located in two places. Classes containing utility methods are stored in the package com.chelpus. 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 patches - This includes libraries, e.g. axml [37] for serializing the AndroidManifest.xml from Android binary into an ASCII formatted, human readable xml and zip4j [69], a Java library to handle ZIP files.
4. The fourth is a modified billing and license library - It is applied in combination with a proxy to redirect inapp billing and licensing calls.

In the resources folder, different predefined custom patches can be found.

Lucky Patcher tries to hide the way it works. The developer uses heavy obfuscation and a flat code structure for the Lucky Patcher core logic. The goal is to make reverse engineering as time consuming as possible. These techniques are applied very effectively since Lucky Patcher itself is attacking such applications. For this reason a blackbox approach is choosen. This analysis focuses on gathering information from the output of a mechanism which should be analysed. Lucky Patcher's different modes are applied on a variety of applications. A lot of data and edge cases is generated this way to identify how Lucky Patcher works.

3.3 Blackbox Analysis of Patched Applications

The blackbox approach is done to identify the parts of the code manipulated by Lucky Patcher and to suggest solutions to protect these weak points. Different applications are used to get a variety of results and see how the different implementations are attacked. The applications are patched using the *Auto Modes*. The approach for creating a modified APK was chosen since .odex files are device specific and cannot be used for a general conclusion of an attack. The outcome is analysed using the methodologies described in section 2.4. Since the only the bytecode is modified, a static analysis is sufficient.
The goal of reverse engineering the code and comparing it to the original application is to discover the changes and analyse them on different abstraction levels. This includes the .dex level, on which Lucky Patcher works, the smali level, which makes the .dex code human readable, as well as Java level, on which the functionality change can be identified. On each level, the modified and original code are compared using diff to retrieve the changes in an easy way as well as ignoring the unchanged code.

Besides circumventing the Google LVL, Lucky Patcher supports the the removing of Amazon's and Samsung's license verification library. A reference application with known source code and an implementation, according to the tutorial of LVL [13], is created. This application is called *LicenseTest*. It gives full control and knowledge of the code and allows the analyse of Lucky Patcher attack on the most basic version. The same application is uploaded with deactivated LVL to the Amazon store and injected with the *Kiwi* 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[78], version 6.3, and Teamspeak 3[92], version 3.0.20.2, for the LVL and A Better Camera [4], 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 N*x* 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. The patching mode is responsible for the patterns applied when patching the LVL. Table 3.1 gives an overview. Amazon and Samsung are always patched the same way. Additional knowledge on the patterns was gained

	Patterns							
Modus	N1	n3	N3	N3i	N4	N5	N6	N7
Auto	X	X	X		X			
Auto (Inversed)	X	X		X	X			
Extreme						X	X	X
Auto+Extreme	X	X	X		X	X	X	X
Auto (Inversed)+Extreme	X	X		X	X	X	X	

Table 3.1: Overview of License Verification Library patching patterns applied by each modus

from the black box analysis.

Before explaining the patterns in detail, additional information has to be provided. In the .dex file analysis, a simplified presentation as *0a* instead of hexadecimal values like *0x0a* is chosen for improved overview in the diff files. When converting .dex files to smali files, the arguments of the opcodes are transfered to variables, e.g. *x* in dex code is v*x* in smali.

When the dex code of an application is modified, the checksum has to be recalculated and the file has to be signed again. This changes can be seen in the diff of the dex files. They are not explicit mentioned in the analysis since they do not change the logic of the code.

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 [20].
In can be seen in the dex code in code snippet 3.1 that *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 variables can be identified as blocks 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, the changes can be seen in their context. 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

The result is the voiding of the *verify()* switch case. 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 custom libraries are attacked as well. An example is AnjLab's inapp billing library [25] of FKUpdater, located at *com/anjlab/android/iab/v3/Security*. The library contains code for the Google in app billing. The pattern is applied to other locations to counter a moved LVL. Similar to pattern N1, pattern N2 targets the *LicenseValidator* class's *verify()* method.
The changes in the .dex file can be seen in code snippet 3.4. The instruction *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

The smali in code snippet 3.5 names the target opcode. Instead of moving the result of the predecessing function to *v5*, the varibale 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 result in the Java code (see code snippet 3.6) is interpreted to more than just the setting of a variable to *true*. Instead of proceeding according to the result of the verification of the signature, the result is ignored and the execution is continued inside the condition. 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

The consequence 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 differentiate between them in the result ouput. They are combined under the same number since both attack the same part of code inside the classes defining the policies. In case of the basic implementation of the LVL, these classes are the *APKExpansionPolicy* and *ServerManagedPolicy* in the *com/google/android/vending/licensing/* folder. Those two classes are examples of policies offered by Google [20]. Pattern N3 attacks their *allowAccess()* method.
In case of pattern N3, *01* is replaced with *11*, while in case of N3i *11* is replaced by *01* (see code snippet 3.7).

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

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

Code Snippet 3.7: Diff on Dex level for N3 pattern

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

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

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

Code Snippet 3.8: Diff on Smali level for N3 pattern

The resulting Java code is shown in code snippet 3.9. The pattern attacks the result of the *allowAccess()* method. While N3 is targeted towards code where the default return value is *false*, pattern N3i is used when the default value is *true*.

```
@@ 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

Both patterns attack the class's *allowAccess()* method which evaluates whether the verification result is according to the policy or not. The return variable is initiated with the denying result. It is changed when the evaluation of the input is successfully verified. Otherwise it stays the same and thus denies the access. The pattern changes the default value to the allowing result. This 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. It is responsible for initializing the license check in its *checkAccess()* method [20].
As seen in code snippet 3.10, it replaces *38* with *33*.

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

Code Snippet 3.10: Diff on Dex level for N4 patch

In the smali code snippet 3.11 this change can be identified as replacing *if-eqz* with *if-ne*. The opcode *if-eqz* takes one argument *v0* while *if-ne* takes two arguments. Since the value is *0* in the .dex file, it is interpreted as *v0* for the second argument.

```
@@ 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 in code snippet 3.11 shows the result of the changes. In the original code, the result of *mPolicy.allow()* was checked. In case the policy did not allow to continue adn thus the result was *false*, the condition block was executed. The changes of the pattern result in the check for inequality of the result of *mPolicy.allow()*. Since the result of the method is the same and thus the condition is never fulfilled, the condition block is never called.

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

Code Snippet 3.12: Diff on Java level for N4 patch

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

Patch Pattern N5

As part of the extreme mode, pattern N5 changes the .dex file similar to pattern N2 and targets the *LicenseValidator*'s *verify()* method. It cannot be applied on the standard implementation of the LVL.
In the Java diff in code snippet codeSnippet:n5DiffJava the result of patching can be seen. The original code stores the parsing result of the first entry of the array *v2_1* in an attribute of an object (*v0_0.a*). After applying the patch, the entry is still parsed but ignored and *v0_0.a* 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

The pattern sets the response code of the object to *LICENSED*. The real reponse code is ignored and the code continues even though the server did not verify the license.

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 the opcode *const/4* consists of the opcode tuple and another tuple of arguments while the original *if-eqz* has the opcode tuple, the argument tuple and a target tuple. In order to get a valid syntax, the code has to be changed accordingly. The result are two *nop* oeprations *00 00*, which are interpreted as *nop* 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 changes of the pattern are presented in code snippet 3.16. Instead for checking the reponse code whether it is *LICENSED* it is set to that value. This results in the constant values inside the if and switch statements. In addition, the check whether the response code is not *NOT_LICENSED* is replaces with the check whether it is not *LICENSED*.

```
@@ Pattern N6 @@
```

```
- if (responseCode != LICENSED || responseCode != NOT_LICENSED ||
    responseCode != LICENSED_OLD_KEY) {
- switch (responseCode) {
+ responseCode = LICENSED;
+ if ((LICENSED != LICENSED) && (LICENSED != LICENSED_OLD_KEY)) {
+ switch (LICENSED) {
```

Code Snippet 3.16: Diff on Java level for N6 patch

This pattern prevents the execution for cases where the *verify()* has to handle reponse codes that are neither *LICENSED*, *NOT_LICENSED* or *LICENSED_OLD_KEY*. It voids the impact of those reponse codes. Instead it proceeds successfully since the reponse code is set to *LICENSED*.

Patch Pattern N7

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

```
@@ Pattern N7 @@
- this.verifyLicense(p7.readInt(), p7.readString(), p7.readString());
+ p7.readInt();
+ this.verifyLicense(0, p7.readString(), p7.readString());
```

Code Snippet 3.17: Diff on Java level for N7 patch

The goal of this attack is to patch the *onTransact*, where ever it may be located. This methods is responsible for calling the *verifyLicense()* implementation of the *ILicenseResultListener*. Instead of passing the reponse code from the server, the method is called using the initialized *0*. This way the server's reponse code is ignored since it is replaced by the code for *LICENSED*.

Amazon Market Patch

Amazon does not have different patches or pattern. Since the *Kiwi* library is injected by Amazon and cannot be customized by the developer, only one patch with pattern is necessary. The pattern is applied twice while patching. The first class tageted is *com/amazon/android/licensing/b.java* while the second class is *com/amazon/android/o/d.java*. While *b.java* is responsible for the verification of the license, *d.java* is reponsible for handling the expiration of the license.
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. The opcode *if-eqz* evaluates only the one argument while *if-ne* takes two arguments. Since the value in the .dex file, where the second argument of *if-ne* is, is *0*, *v0* and *v0* are compared.

```
@@ 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 presentation helps to interpret the changes of the attack. Since the if statement is now always wrong inside the *b.java* class, the code block for repsonse codes not *APPLICATION_LICENSE* is never called. The same canges are applied to the d.java class. 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 new check is always false as in *b.java*.

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

Code Snippet 3.20: Diff on Java level for Amazon patch

The analysis of the Amazon patch indicates that there are less patterns needed since code modifications have to be expected. Patches are applied to manipulate the checks in

case the response code is different than *APPLICATION_LICENSE*. 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. For this reason cracking the library requires only one patch. The patch is applied on the *LicenseRetriever* and *Zirconia* class in the *com/samsung/zirconia* package. The Samsung patch uses two patterns, called S1 and S2 in order to distinguish between them. While S1 is applied on both classes once, 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 now uses *v0* and *v0*. The result of this comparison is always *true*. Pattern S2 has the effect that *v0* does not return the result of the preceding 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 presentation in code snippet 3.23 contains the changes in Java code. 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

The result of applying the patch is that not only the license file checks are voided and return the verification as *true* on default. In addition, 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

The summary of the LVL patterns and their use in the patching modes can be seen in table 3.1. Amazon and Samsung are always successful since not the developer implementation is attacked but the library itself, which is always the same.
When patching Amazon, Samsung and the LVL with the auto and inversed auto mode, the patches are applied determined to the important parts of the application. They are effective 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 files in other locations 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.

Table 3.2 gives an overview of patched applications. Lucky Patcher does not guarantee to be successful with one or more patching modes.
The first example is *LicenseTest*. While circumventing the license verification with the auto mode alone and combined with the extreme mode is possible, it does not work with the other modes. One reason is due to the inversed auto mode tailored to the opposed configuration of the *allowAccess()* method implemented. Another reason is

	Application		
Modus	LicenseTest	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

that the extreme mode is an addition to counter modified license verification implementations.
The second example is *Runtastic Pro*. This implementation is altered in a bad way which makes it vulnerable to all patching modes. In the pirated and unmodified application, the user is told that the application is under maintenance. When the patches are applied, the user can login as if the application was bought in the store.
The third example is *Teamspeak 3*. Their interpretation for the license verification is able to withstand all attacks.

Lucky Patcher is carrying out the attack by modifying the logic of the license verification libraries. The circumvention of the verification process is achieved by apply only minor changes. Single opcodes are changed to alter checks. The resulting code evaluates a constant with desired outcome instead of the result of the initial method. This way bad response codes can be ignored while the outcome is enforced. It is very difficult to avoid this kind of attack since the license verification process is dependent on the binary *yes/no* evaluation. An abstraction of the unary verification mechanism is presented in figure 2.12.

The changes are applied by extracting, modifying and repacking the *class.dex* similar to the build process of subsection 2.2.1. The modification of the file entails that its checksum and signature have to be recalculated. Since Android can detect tampering of the APK, the digest of the file in the *META-INF* folder and its manifest files has to be adjusted as well [98]. Lucky Patcher does not have the developer's private key to sign the files accordingly. This causes problems when certificate is checked for its origin but since Android allows self-signed certificates it does not matter [48]. Android just compares the certificates in case an application is updated with a APK containing the same package name or when trust relationships between applications are about to be established. [98] The only limitation is the installation of the modified APK is only

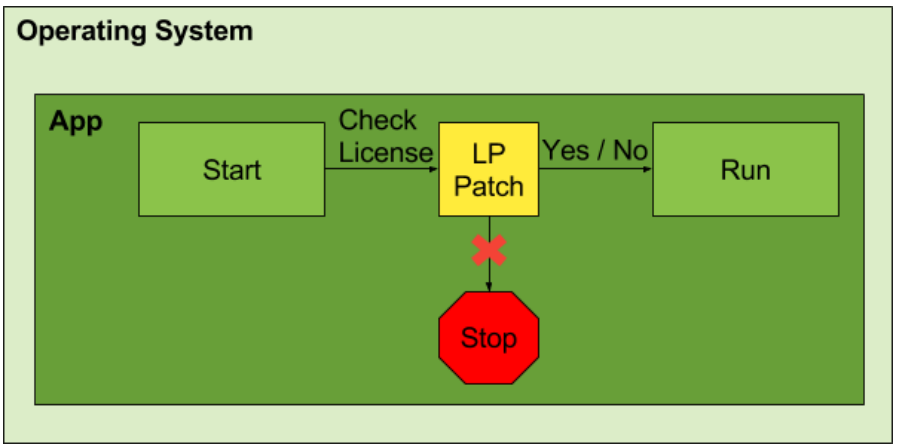


Figure 3.2: Abstraction of the current attack on the license verification mechanism

possible when the original application is removed and future updates are not possible as long as the new APK contains the same certificate.
The creation of the modified APK does not require *root* and the cracked APK can be distributed without limitations. This makes piracy easy for users and is a huge threat to developers.

4 Countermeasures for Developers

Now that that the methodology of Lucky Patcher's attacks is analyzed, suggestions to prevent the circumvention of the license verification are 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 advanced attacks.
The second chapter proposes a content driven approach with additional encryption to leave the license verification. /newline The third chapter takes an outlook on ART to tackle the shortcomings of dex bytecode regarding the license verification.

4.1 License Verification Library Extension

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

4.1.1 Modifications on the Google LVL

Attacking with Lucky Patcher is often successful because many developers do not customize the LVL at all. One reason is the protection against casual piracy. Using the basic implementation prevents the use after copying of the APK from one device to another. This is only effective against uninformed attackers but sufficient for the developer. Another reason is that they do not know where exactly to fortify the library and thus they do not want to spend additional effort in gathering information. [60] [71]
This thesis presents two approaches to fortify an application against attacks by Lucky Patcher.
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 is not targeted by Lucky Patcher's auto patching modes.

While the idea of the approaches is valid for all implementations, these approaches are only implementable when using LVL since they requires access to the source code.

Modify the Library

Lucky Patcher's attack is reliant on successfully applying the patching patterns. For this reason, going actively against these patterns should always be the first step to challenge Lucky Patcher. Modifying and improving the LVL does not only protect from patterns. Increasing complexity of the application's bytecode makes it unique and harder to reengineer. [60]
There are three areas the developer should focus on when modifying the LVL [60].

1. core licensing library logic
2. entry and exit 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 and N6)
- move the LVL package into the application (see patterns N2 and N7)
- use additional threads to handle different steps in the license verification process
- implement functions inline where possible (see patterns N2, N5 and 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. [60]
The entry and exit points can be attacked by creating a counterfeit version of the LVL that implements same interface. An unique 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. [60]
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. [60]
These modifications are easy to apply and make the implementation unique. There are unlimited ways to implement them. This makes it hard to attack automatically with an universal solution. 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 find a weakness in the code. The gained knowledge can be used to create a custom patch cracking the application. 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. [60]

Native Implementation

Lucky Patcher's automatic patching modes, as described in chapter 3, target the application's bytecode inside the *classes.dex*. Since Android supports the Native Development Kit (NDK), parts of the application can be implemented using native code.
Usually the NDK targets CPU intensive tasks, such as game engines and signal processing, but it can be used for any other purposes as well. Google suggests using it only if necessary since it increases the complexity of an application [14]. This is a desired side effect when implementing the LVL natively. Native code, in opposite of byte-code, does not contain much meta-data, such as local variable types or class structure. Its information is discarded on compilation and thus the code is harder to understand.
There are two scenarios for creating a native implementation of the LVL.

- The developers implements an its own native version of the LVL
- Google provides a native implementation of the LVL

In the first scenario, the implementation is the developer's responsibility. When the developer implements its interpretation 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 process and its implementation into the application itself. First they need to reverse engineering the native code, then they can start with searching for a way to break the license verification. Only if they succeed in these two steps, they can repack the cracked native library and make it available as a custom patch for Lucky Patcher. This scares off attackers since the circumventing of the native license verification requires a lot of knowledge and time. As long as the attacker has to evaluate the workload with the gain of cracking the application, it is considered a safe method, implied the developer has enough available resources. [71]
In the second scenario, Google is responsible for delivering the implementation. Instead of providing Java code, Google could provide a native version of the LVL. In the beginning, it is be harder to find vulnerabilities than it was with the Java version for which the source code is provided. It takes some time for the attackers to reengineer and crack the library. The additional effort is justified for the attackers since the library is implemented into many applications of the Play Store. After a while this license verification faces the same problem as Amazon's or Samsung's libraries. Having only one custom patch applied by Lucky Patcher would be able to crack all applications universally. For this reason, the implementation must include two essential features.

- heavy obfuscation and encryption must be applied
- dynamic code generation and automatic customization every time it is loaded - 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 one has come up with such universal approach for the broad masses. This indicates that still a lot of research and work has to be done to implement this solution. As long as the the license verification library is implemented in native code, the automatic patching modes of Lucky Patcher do not work since they only target dex bytecode. Attackers have to crack the native library and offer it as a custom patch.

4.1.2 Tampering Protection

The unmodified execution is only guaranteed as long as the integrity of the environment is ensured.

When circumventing the LVL the code has to be modified. There are different indicators for an attack on the application. Breaches of integrity are forced debuggability, *root*, an installed cracking applications or installation from a soruce different than the store.
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

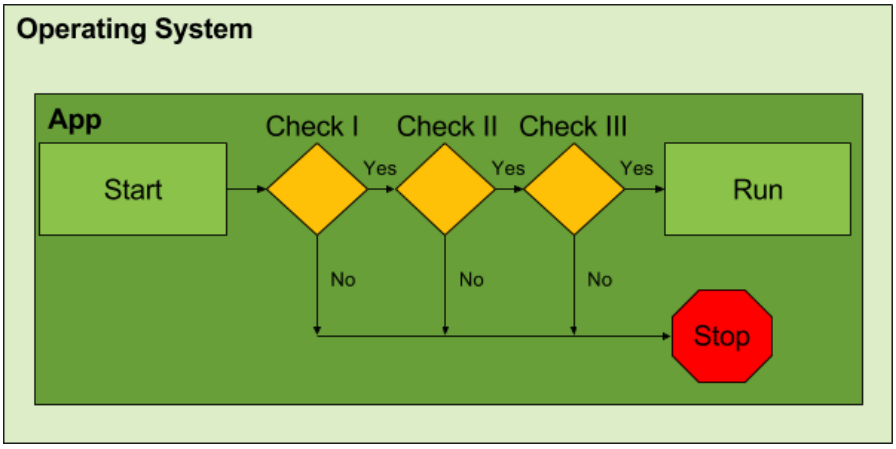


Figure 4.1: Introduction of additional tests to check environment and integrity of the application

countermeasures have the *yes/no* schema, they can be circumvented easily. The goal of these checks is to increase the workload of an attacker. The code has to be analysed in order to find, understand and patch them. The checks can be spread inside the application to unexpectedly crash the application. The attacker not only has to invest time to figure out why the application crashes randomly but also to find these checks. Obfuscated and implementating it randomly increases the effort as well. 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 [18]. This features are used to gain information about the flow of the app and to reenginer functionality. With the results of this analysis weak points are identified and custom patches are developed.
The debug flag is not set in release builds on the application stores, but it can be activated by chaning it in the *classes.dex*. 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 can be used to alter applications or extract protected data. The developer can check whether *root* is available on the device and eventually exclude users which use it. The developer needs 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 [64], which is said to be used in security critical applications like Android Pay [16] [77].
Since *root* is achieved by the *su* file in the filesystem, the application can search for its existance in the common locations. In case the search is successful, the execution of the application can be terminated.

Lucky Patcher

Having Lucky Patcher installed is a astrong indicator that the user is pirating applications. It can be extended to detect additional unwanted applications by adding their package name to the check [100]. The check is more specific as the *root* check as it only excludes people who have a piracy tool.
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.

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

Code Snippet 4.2: Example code for checking for root

Sideload

APKs can be cracked on a different device, 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. Some custom Android versions already include a library called *AntiPiracySupport* [40]. It has a similar goal and blacklists and disables pirated applications.
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. In case he does not, the application will not work when retrieved from a not listed 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 [60].

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

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

Signature

Application code is signed to authenticate an application developer and enable him to provide updates for the application [98]. Since the signature has to be rewritten when cracking the application, it is used as an indicator for attacks [87].
The approach is similar to Google Maps inside an application. When launching the map the applications sends the SHA1 signature and the API key to the server which verifies whether the application is allowed to display the map [54].
The code for the signature check can be seen in code snippet 4.5. In order to check the application's signature, the original signature has to be provided (see line 53). The application's signature fetched from the package information (line 57ff). In case the signature cannot be retrieved or the signature do not match, the check terminates the application.

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

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

4.1.3 Obfuscation

The first steps to fortify the LVL have been made. The library is modified and the environment's integrity is checked. This helps against Lucky Patcher's automatic patching modes but is still vulnerable to manual attacks. Android applications are at high risk of being reverse engineered as it is much easier to decompile the simple bytecode over native code. In order to hinder reverse engineering and prevent the development of custom patches, obfuscation is introduced.
The obfuscator is an easy to apply protection and should be used in every application. It does not protect against automated attacks since it does not alter the program itself. Obfuscation can be applied to the standard version of the LVL as well but it is no protection since the source code is known. Its full potential is unleashed when combined with the unique implementation. The goal is to make the attackers work much more time consuming up to the point where the effort is no longer profitable. When the

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

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

attacker does not have proper class and method naming it is harder to identify the purpose of the particular part which is analysed. It makes it much harder to develop a custom patch. [60]
The obfuscator can either be applied to the source code or bytecode. There are open-source and commercial Java obfuscators available that are also working on Android, e.g. *ProGuard* [66]. It is applied in an addtional step in the build process. The .class files are transformed right after the Java compiler and before *dx* converts to the *classes.dex*. [66] [21] Some dex obfuscators exist as well, like *DexProtector* [44]. These obfuscators are applied after the *classes.dex* has been created.
Three obfuscators are explained in detail. will be explained in the following. [60]

There are limitations since certain methods cannot be obfuscated. They rely on the Android framework, e.g. *onCreate()* which has to callable by the Android system. The developer should avoid implementing license verification related code inside these methods since attackers will look into these methods. [60]

Applying obfuscators does not directly protect from Lucky Patcher. When the implementation of the LVL is unique the analysis is much more time consuming and thus provides an additional protection layer for the application. It forces attackers to invest more effort in order to understand the application and thus reduces the likelihood of attackers targeting the application.

Proguard

The most basic obfuscation tool is *ProGuard*. It is an open-source Java obfuscator which is part of the Android SDK and free of charge.
ProGuards's 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

removes unused classes, fields, methods and attributes which got past javac shrinks, optimizes and obfuscates java .class files optimization step methods are inlined, unused parameters removed, classes and methods made private/static/final as possible optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code variable identifiers name scrambling for packages,classes, methods and fields obfuscation step name and identifiers mangled, data obfuscation is performed, packages flattened, methods renamed to same name and overloading differentiates them lexical-sorted strings like a, b, c, ..., aa, ab, original identifiers give information about interesting parts of a program merges identical code blocks performs optimiztations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying

string obfuscationa, string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog, deobfuscation stub

shrinks the code size by automatically removing unused classes, detects and highlights dead code, but leaves the developer to remove it manually

advantage of minimizing the memory usage, e development process in step ”a” or step ”b”
- 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' [81] [50] [95] [66] [21]

Dexguard

DexGuard is a commercial Android obfuscator for Java code. It is considered the son of *ProGuard* and specialized on Android. The provided methods are a super set of *ProGuard*'s ones. [95]
In addition to the obfuscation, *DexGuard* offers application and platform integrity, resource protection, communication hardening and code protection [57]. This way, the application is not only fortified against code analysis but also steps against the reverse engineering itself are applied.
The side effect of using *DexGuard* is that the application becomes smaller and the performance is increased by the smaller memory usage.

Allatori

Allatori is acommercial Android obfuscator from Smardec. Similar to *DexGuard* it provides a superset of *ProGuard*'s methods. [2]
The features of *Allatori* include name obfuscation, string encryption and debug information obfuscation. Even flattening of the structure and obfuscationof the control flow are possible. Another feature is the addition of complexity to algorithms, e.g. loops are modified in a way that reverse engineering tool do not recognize them as such. [95] [3]
Overall, the resulting application has a decreased .dex file and memory footprint while having increased speed.

4.2 Content Driven Application

Android bytecode can easily be decompiled and altered to crack specific mechanisms. Content driven applications are introduced to protect the important parts of an application and prevent piracy.
In addition to the basic content driven application, possibilities to apply encryption are presented.

4.2.1 Content Server

The first approach to fight the shortcomings of the license verification libraries by moving it to a server. 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's .dex file, Lucky Patcher is not able to manipulate it anymore.
The implementation can be described with the application Spotify [85] as a reference.

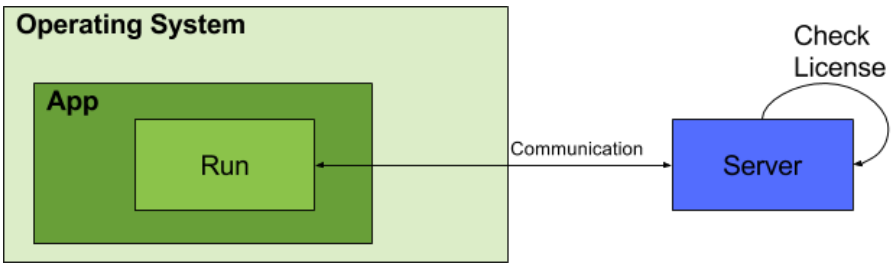


Figure 4.2: Abstraction of an application and a content server

Instead of verifying the license locally on the device, the user has to enter his credentials and send them 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 login process inside the application by manipulating the code. Since the content is on the server and the user has to be authorized on it, no content is available inside the application. Thus attacks on the applications itself do not work anymore by flipping checks.

In general, a content server is good solution against piracy but it has downsides as well. The first problem is that this architecture cannot be applied to all business models. This 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 money is needed for the server, but an additional application for the server has to be created as well. Not every developer can handle this additional workload.
The third problem is the resulting always online necessity which limits the freedom of users and creates additional unnecessary traffic. This is not accepted by all users.

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

4.2.2 Encryption

Since not all application suit the server client model, encryption is introduced as another countermeasure to prevent piracy. Encryption has two advantages in the fight against cracking applications since it is more complex. The first advantage is that the cryptographic keys are not predictable. Lucky Patcher is not able to patch the application in a way that a certain outcome is enforced. The second advantage is that the application does not work in the way it is intended when the decryption key is not present or the decryption methods are patched.

Encryption

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

Encryption - Resources

The first approach is to apply encryption on the application's static resources. This can include the application's hard coded strings or 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 will not understand the application because all strings or images are still encrypted and have no meaning.
Figure 4.3 shows the abstract implementation of resource decryption.

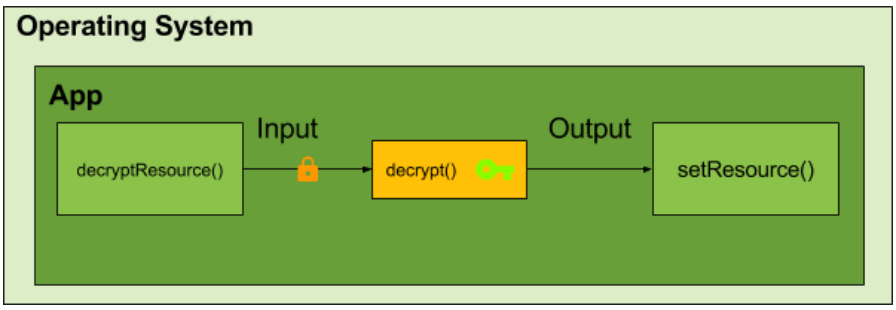


Figure 4.3: Encrypted resources have to be decrypted before they are used or displayed

Encryption - Action Obfuscator

The second approach is to use encryption as obfuscation. The idea is to have a single method to deligate all other method calls according to an encrypted parameter.
When an attacker does a static analysis of the code, the links between method call and executing method are not apparent. It forces the attacker to use a dynamic analysis method instead. The fortification of the mechanism is improved when encrypted arguments are passed as well and the decryption is done in the executing method. This requires more than just opcodes to be circumvented.
An abstract presentation of the mechanism can be seen in figure 4.4.

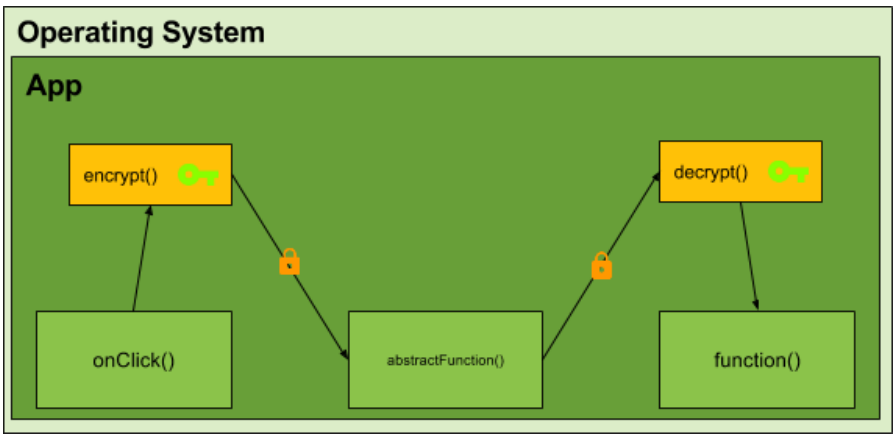


Figure 4.4: Encrypted actions to obfuscate dependencies

Encryption - Communication

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

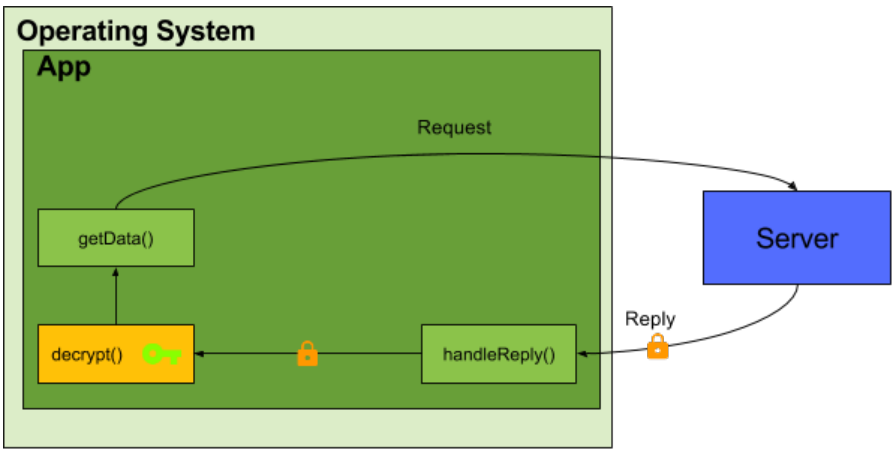


Figure 4.5: Encrypted communication with a server

Key Storage

Decryption does not work without a decryption key. This key has to be stored in a secure place since the encryption is only as strong as the protection of the key.

Key Storage - Online and Caching

The first approach 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.
On decrypt method call, the application tries to retrieve a cached cryptographic key. In case a cached key is available, the application starts to decrypt the content. Otherwise the key is requested from the server. The server does a verification of the user similar to the license verification libraries. In case the check is successful, the decryption key is send to the device instead of a simple yes or no. The advantage over having the original implementation is that the key can neither be accessed without the verification on the server and nor guessed by an attacker to circumvent this countermeasure.
The key can either be retrieved from the server for each decryption action or it can be cached on the device, similar to the license verification policy. Caching should be favored since getting the key for each action not only requires to be online but slows down the application and generates additional traffic.
In order to improve security, keys can be changed when updating the version of the application or be user specific.

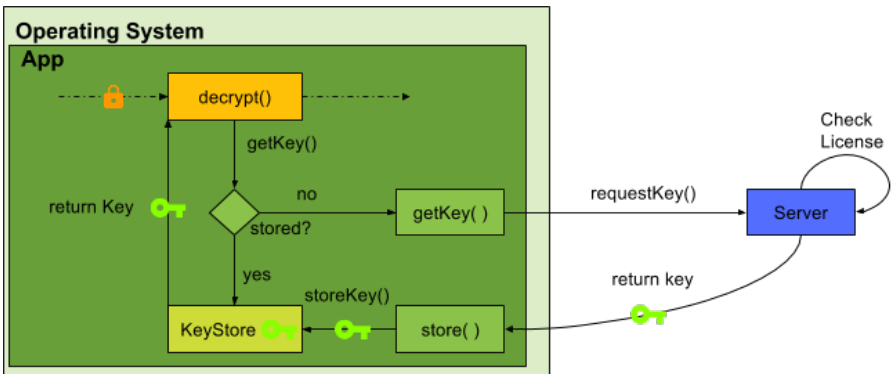


Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

Key Storage - Secure Element

Since there are possibilities to read the cached key [93] and crack the encryption this way, the use of a Secure Element (SE) is proposed.
A secure element is a tamper-resistant platform which can be used to securely host applications and cryptographic keys [52]. There are different form factors for SEs [52]. For Android, the microSD is for Android the form factor of choice. It can be either mounted in the microSD card slot or on the USB interface by using an adapter. Using the USB interface requires the device to support USB On-The-Go (OTG) [97]. The SE is accessed over reads and writes to its filesystem. Since the SE has to be small to fit the size of an microSD card and is powered by the host system, its hardware capabilities are constrained. The result is a performance as low as 25MHz. This does not allow complex computations on the SE. [86] 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.
At the moment, the integration of a SE comes with some problems.

- the user has to buy extra hardware
- not all devices have a microSD card slot or support OTG
- no unified implementations for communication with the SE across manufacturers

The first problem is that the user has to buy extra hardware. This means extra money has to be spend and the hardware has to be always around. In addition, the connection

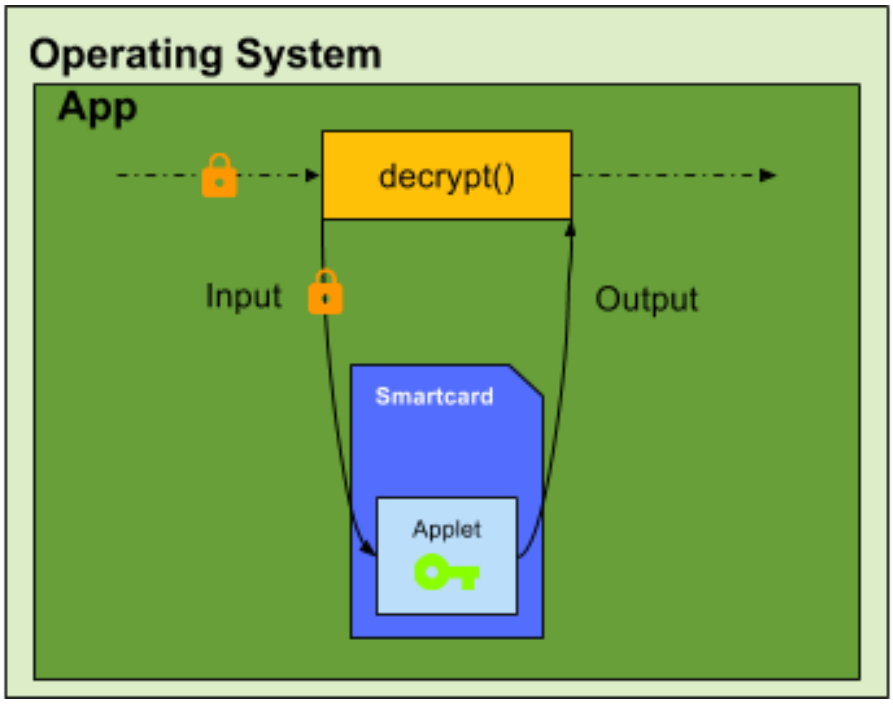


Figure 4.7: Decryption by using a smartcard

to the device using a cable is not the most convenient solution. /newline The second problem is that some devices neither have an microSD card slot nor implemented 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 [82].

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?

4.2.3 Key Storage - Trusted Execution Environment

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

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

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

WAS IST ES? WAS MACHT ES? WIE IMPLEMENTIERT MAN? `https://source.android.com/security/trusty/index.html` promoted as be all end all solution for mobile security in theory isolated processing core with isolated memory, cannot be influenced by the outside and runs with priviliged acces allows secure processing in the "secure world" that the "Normal world" cannot influence or beware of senisitve processing offloaded to protect information from malware

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

example Trustzone, Knox

[35][29]

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

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

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

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

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

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

[35][29]

not accessible to anybody in general many different solutions, focus should be on one unique standard in order to fix the problems and make compatibility , google already started by integrating features of samsung's knox into android lollipop [79]

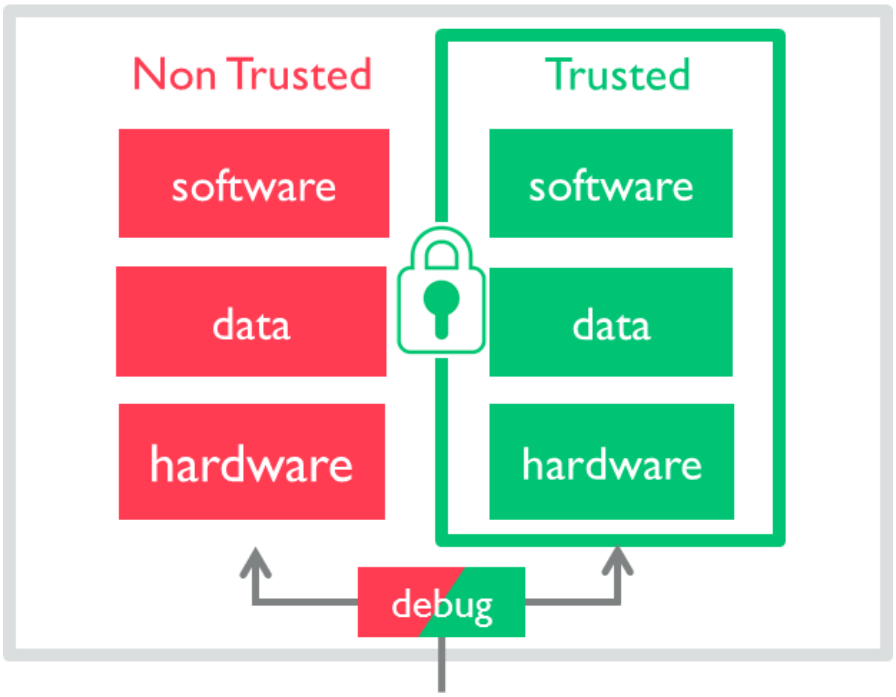


Figure 4.8: tee [29]

4.2.4 Key Management

4.3 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
[67]

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
[67]

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

as long as self signing is allowed, applications can be changes, signed and installed, but google does not want a walled garden as apple on iOS, allowing only applications they approve[48] [98] walled garden

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 [71] 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 [81] 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 [62]

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[71], 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

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

von Lucky Patcher. Im Falle, dass ein Developer "Sicherheit" vorzieht und seine App in einem alternativen Store anbietet, gibt es zwei 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. Nur weil ein kopeirschutz nicht gekackt ist heißt es noch ncith dass er nciht knackbar ist, sobald genügend interesse besteht wird es jemand versuchen

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 [71]

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

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

[6] 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).

[7] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] Android Developers. *ProGuard*. URL: http://developer.android.com/tools/help/proguard.html (visited on 03/07/2016).

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

[30] P. Bernhard. "A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks." Master's Thesis. Technische Universität München, Fakultät für Informatik, Aug. 2015.

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

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

[33] 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).

[34] 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).

[35] J. T. Charles Holmes. *An Infestation of Dragons - Exploring Vulnerabilities in the ARM TrustZone Architecture*. URL: https://usmile.at/sites/default/files/androidsecuritysymposium/presentations/Thomas_Holmes_AnInfestationOfDragons.pdf (visited on 01/24/2016).

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

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

[38] 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).

[39] 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).

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

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

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

[43] 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).

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

[45] 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).

[46] 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).

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

[48] N. Elenkov. *Android code signing*. URL: http://nelenkov.blogspot.de/2013/04/android-code-signing.html (visited on 03/04/2016).

[49] 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).

[50] 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).

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

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

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

[54] Google Developers. *Get API Key*. URL: https://developers.google.com/maps/documentation/android-api/signup?hl=de#display_your_apps_certificate_information (visited on 03/06/2016).

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

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

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

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

[59] J. Jang, H. Ji, J. Hong, J. Jung, D. Kim, and S. K. Jung. "Protecting Android Applications with Steganography-based Software Watermarking." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1657–1658. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480673.

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

[61] 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).

[62] 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).

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

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

[65] 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).

[66] E. Lafortune. *ProGuard*. URL: https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html#manual/introduction.html (visited on 03/07/2016).

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

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

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

[70] 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).

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

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

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

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

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

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

[77] 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).

[78] 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).

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

[80] 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).

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

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

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

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

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

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

[87] stack overflow. *android detect LVL removal*. URL: http://stackoverflow.com/questions/19658890/android-detect-lvl-removal (visited on 03/06/2016).

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

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

[90] 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).

[91] 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).

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

[93] 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).

[94] 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).

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

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

[97] USB Implementers Forum, Inc. *USB On-The-Go and Embedded Host*. URL: http://www.usb.org/developers/onthego (visited on 02/29/2016).

[98] W. Verduzco. *Android Signature Verification Basics*. URL: http://www.xda-developers.com/application-signature-verification-how-it-works-how-to-disable-it-with-xposed-and-why-you-shouldnt/ (visited on 03/04/2016).

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

[100] 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).