DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations into 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 into 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, 2016

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, 2016 Johannes Neutze, B. Sc.

Assumptions

This thesis assumes basic knowledge of informatics and modern computing. When describing Android or other general concepts, only the aspects relevant to this work are written. When proposing solutions, it is understood that each of them has their own restrictions and vulnerabilities, which are not followed through to the end.

Abstract

Android is target to massive levels of attack, stealing intellectual property. Huge losses occur through general attacks on license verification mechanisms. Lucky Patcher is recognized as one of the major culprits. The scope of this work is to understand Lucky Patcher's strategy of attacks and propose countermeasures. A black box analysis is chosen and an exhaustive test done, attacking several applications. The strategy is made apparent and traced back to the single concept of a unary decision. An array of countermeasures is proposed, some fortifying the current basis and others suggesting disruptive change. All proposed solutions exist in the trade off between security, openness and complexity.

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 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.

APK 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.

Kiwi The name of Amazon's license verification library.

Lucky Patcher Android cracking tool used to void license verification mechanisms in premium Android applications.

Zirconia The name of Samsung's license verification library.

Acronyms

.dex Dalvik EXecutable file.

.jar Java Archive.

.odex Optimized Dalvik EXecutable file.

ADB Android Debug Bridge.

API Application Programming Interface.

APK Android Application Package.

ART Android RunTime.

DRM Digital Rights Management.

DVM Dalvik Virtual Machine.

ELF Extensible Linking Format.

IP Intellectual Property.

JIT Just-In-Time.

JNI Java Native Interface.

JVM Java 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 Intellectual Property (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 [73].
Software licenses come in different variants. They range from open source, over usage for a limited time, to usage of a limited set of features. Since using 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 [74].

1.2 Motivation

Licensing is also present in Android. With a market share of almost 82.8% in Q2 of 2015 [46], it is the most used mobile Operating System (OS). According to Google, over 1.4 billion active devices in the last 30 days in September 2015 [26]. This giant number of Android devices is powered by Google Play [44]. 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 [77]. 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 [52].

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. They were ported to Android as an afterthought. Now, with Android's overwhelming market share, they focus heavily on Android [62]. But this also creates a downside. The expanding market for Android, offering many high quality applications, draws the attention of software pirates. Crackers bypass license mechanisms of applications and offer them for free or use them to distribute malware. Cracked applications can redirect cash flows as will be seen in subsection 2.1.1, which is a lucrative business model.
Android developers are aware of the situation [78] and express their need to protect their IP on platforms like xda-developers [67] or stackoverflow [71]. Many of the developers have problems with the license verification mechanism and name Lucky Patcher as one of their biggest problems [72].

The scope of this thesis is the analysis of the Android cracking application Lucky Patcher and the suggestion of countermeasures for developers.

1.3 Related Work

Lucky Patcher and license verification have already been the topic of scientific work.
In his master's thesis *A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks* [22] Bernhard takes a look at license verification and 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* [57] 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.

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 application analysis are introduced.

2.1 Software Piracy

According to Apple, $11 billion are lost each year due to piracy [21]. Software piracy is defined as the unauthorized reproduction, distribution and selling of software [21]. 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 application 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.
The first type is in-app purchases. They are a popular source of income for so called freemium or lite versions of application lications. 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. Commissions earned this way are assigned according to the Ad Unit ID [41] stored in the application. When an application is pirated, this ID can be replaced by the pirate's ID and commissions funneled to the pirate.

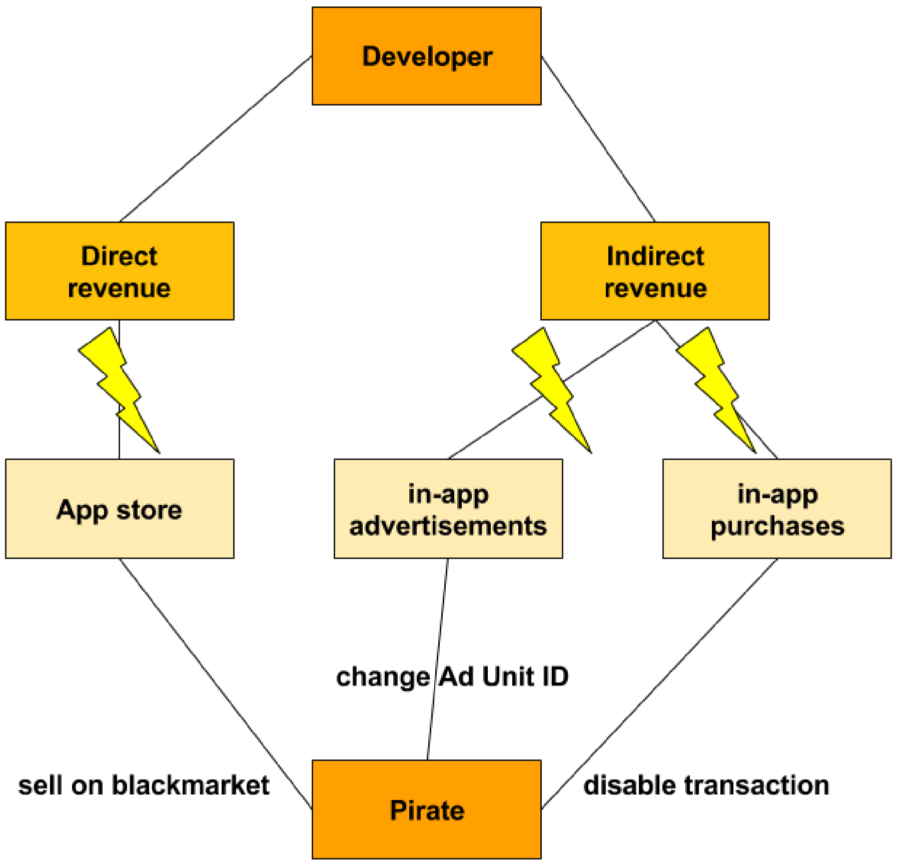


Figure 2.1: Different ways to generate revenue and how the pirate can cut them

Beside monetary issues, additional problems arise when the application is moved to a black market store or website and then distributed without the environment of an official application 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 behavior to the developer. This results in the loss of reputation and potential future revenues not even connected to original 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 [55].
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 skills 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 being aware of it. 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. [25] [55]
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 [35]. 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 the user to black market applications, like Blackmart [23], and websites offering cracked Android Application Package (APK), such as crackApk [31]. 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 [20]. 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. [67] [78] 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 [48].
Android applications are at an especially high risk for piracy because of their use of bytecode, which is an easy target of 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.

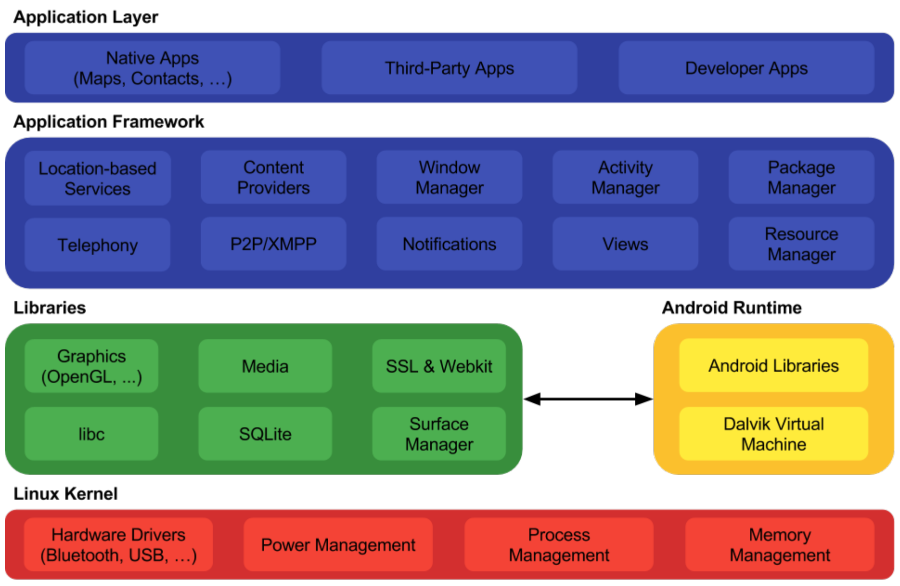


Figure 2.2: Android's architecture [24]

Figure 2.2 gives an overview of Android's architecture.
The basis of the system is its kernel. It is responsible for power and memory management and controls the device drivers.

The layer above the kernel contains the native libraries of the system and the Android Runtime. [1]
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.
This software stack allows Android to run on a wide range of devices with different hardware and offers a simple environment to run applications.

This chapter outlines the aspects of Android, needed to understand the analysis and proposals.

2.2.1 Android Application Package

Android applications are distributed and installed using the APK file format. The APK format is based on the ZIP file archive format and contains the code and resources of the application. They can be installed using different mechanisms. The prefered and most secure is the installation from the Google Play Store. It provides the APK from a trusted source and installs it automatically. Other application stores provide the APK from a trusted source and facilitate the manual installation. Yet other application stores are less trusted. In addition, the APK can be obtained from anywhere and installed manually. The build process of an APK contains several steps which are visualized in figure 2.3.

Since Android applications are usually written in Java, there are similarities to the Java program build process. Upon compilation, the source code is transformed into .class files by the Java Compiler javac. Each Java class is stored as bytecode in the corresponding .class file. Java bytecode can recompiled to become readable. To prevent that, obfuscation can be applied as described in section 4.1.4. When all Java classes are compiled to .class files, they are packed into a Java Archive (.jar) file.
Android uses a Virtual Machine (VM) different from Java. It requires the Java bytecode to be converted to a different format - the 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.

[1]While Android RunTime (ART) and Dalvik Virtual Machine (DVM) are acronyms, they are also used to denote two different generations of runtime as will be described in section 2.2.4 and in section 2.2.5

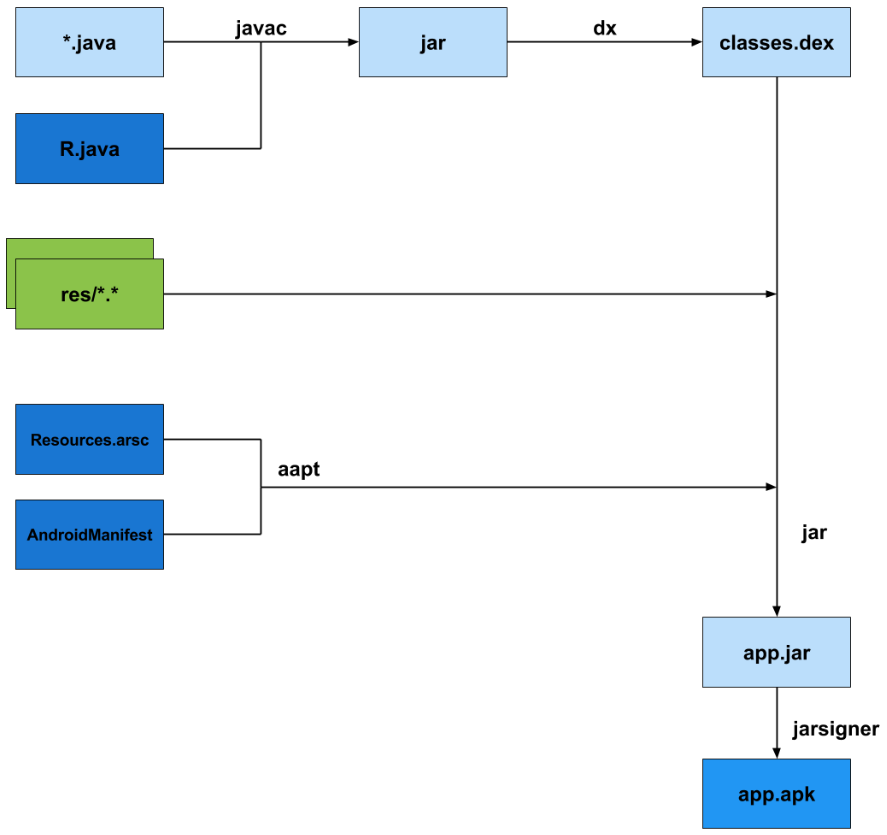


Figure 2.3: APK build process [54]

The APK itself consists of three parts.

- *classes.dex*, a file containing the bytecode
- resource files (*res/*.**), a directory containing static content like images, the strings.xml and the layout.xml files
- *Resources.arsc*, containing compiled resources, and *AndroidManifest.xml*, providing essential information like required permissions

The *apkBuilder* combines these files into one archive file, which is signed and zipaligned to become a valid APK. The *jarsigner* uses the developer's private key to sign the application similar to the Java code signing [37]. This signing process ensures integrity and authenticity using a self-signed certificate, i.e. it provides a key usable to decrypt and authenticates the entity it claims to come from. As there is no certificate authority

(CA), this entity cannot be linked to a person or entity in the real world. Upon update the entities previous and current version can be compared and the update prevented on mismatch. [58] Afterwards *zipalign* is used to mark uncompressed data. [17] [80] [54]
The structure of a final APK file has the following content as minimum as seen in figure 2.4.

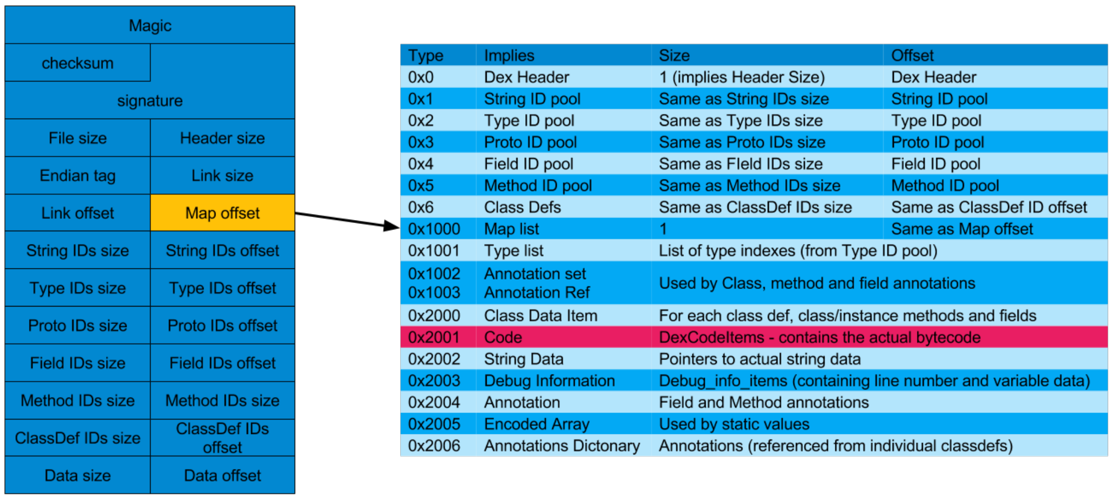
```
APK
|
|--
AndroidManifest.xml
|-- META-INF
|   |-- CERT.RSA
|   |-- MANIFEST.SF
|   `-- MANIFEST.MF
|-- classes.dex
|-- res
|   |-- drawable
|   |   `-- icon.png
|   `-- layout
|       `-- main.xml
`-- resources.arsc
```

Figure 2.4: APK folder structure

The *AndroidManifest.xml* and the *classes.dex* have already been covered.
The APK has the same code signing mechanism as Java . The *META-INF* folder is used to store the signing information, e.g. the manifest and certificate. [37] [59]
While the static resources, like drawables and layouts, are in the res folder, the resources.arsc contains the compiled resources.
Native libraries 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). They are stored as **.so* files in the libs folder, sorted by their specific architecture, like armeabi-v7a for ARM or x86 for Intel processors. [50] [36]

2.2.2 Dalvik Executable File Format

As explained in subsection 2.2.1, an Android application is distributed using APKs. The APK contains the Dalvik executable file *classes.dex*. Figure 2.5 shows the structure of this file. First, the actual bytecode marked red in figure 2.5 is explained, then the checksum and signature mechanism.
The bytecode is a sequence of instructions consisting of an opcode and a number of arguments. The opcode is 8 bit length and determines number and interpretation, e.g. boolean, constant or variable, of its arguments. Instructions are of variable length and aligned to multiples of 16 bit. Instead of using actual data as arguments, references to Dalvik registers are used. [7] [60]



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

Figure 2.5: .dex file format [54]

In the context of this work, the most important parts of the header are the checksum and the signature. The checksum field contains the Adler32 checksum of the .dex file. It is calculated from all fields of the file except the magic field and itself. The checksum is used to detect whether the file is corrupt. The signature field contains the SHA-1 hash value of the file minus the magic field, checksum field and itself. The file can be uniquely identified by the signature, which is also part of the code signing stored in the manifest files to ensure integrity and authenticity. When the .dex file is modified, both the checksum and signature have to be recalculated and updated. [12] [36]
Dex bytecode can be optimized. Upon installation, improvements specific to the underlying 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, Dalvik bytecode can be reverse engineered back to Java source code. As it contains a lot of meta information, e.g. variable and method names, the resulting Java code is reasonably easy to understand, exposing the application logic.

2.2.3 Application Installation

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 2.6). The differences will be explained in the following subsections.

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 *CERT.RSA* contains the public key to decrypt the *MANIFEST.SF* and compare it to the *MANIFEST.MF*. The manifest *MANIFEST.MF* contains the complete list of files and their signatures of the APK, e.g. *classes.dex*. While the manifest files are used to detect tampering, the certificate is used to identify the code signer but as said before only in an abstract way that cannot be linked to a real world entity. Since Android allows self-signed certificates, there is no authority defining right or wrong, e.g. if an application is authentic or a signing developer is a pirate. [37] [80] [58]
The installation can be performed in two ways depending on the runtime of the Android OS. For 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, an .odex file is an optimization tailored for optimal performance on a specific device performed on installation.
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 in section 2.2.5.

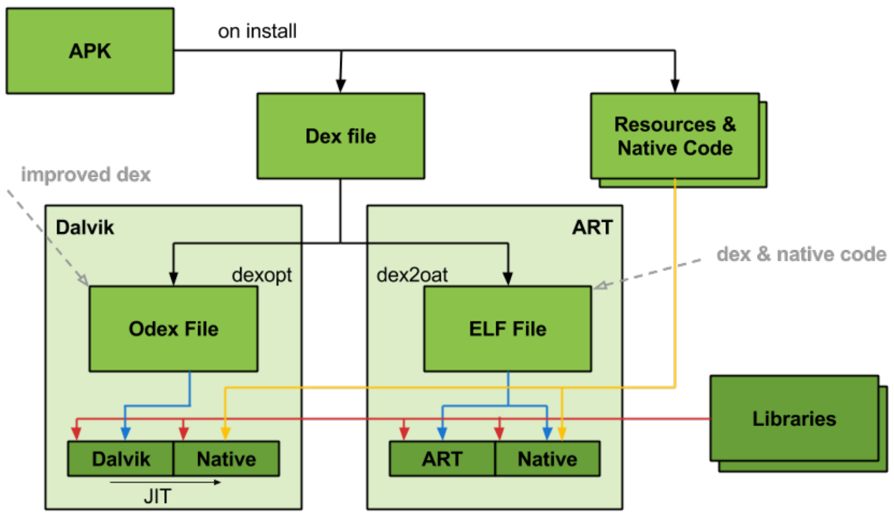


Figure 2.6: Installing an APK on a device [19]

After the bytecode is optimized respectively compiled, the application can be run. When run on the device, Android creates a sandboxed environment for this application only.

2.2.4 DVM - Dalvik Virtual Machine

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 [18].
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 space 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, the DVM 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 needs less instructions for execution than stack-based architecture, which results in less CPU cycles and thus less power consumption. 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 for instructions. In addition to the lower level changes in the DVM, e.g. optimizations for memory sharing between different processes [38], Android uses special concepts, like zygotes. [36] [54]
The last big change made to the DVM was the introduction of Just-In-Time (JIT) compilation, which will be part of the discussion in subsection 2.2.5, in Android version 2.2 *Froyo*.

2.2.5 ART - Android Runtime

In Android version 4.4 *Kitkat* Google introduced ART, designed to replace the DVM. Among other things, it does away with .dex files and replaces them with native code. This refutes all attempts to attack the reengineerable dex bytecode. For compatibility reasons, APKs still have to provide the *classes.dex*. With the release of version 5.0 *Lollipop* ART became the runtime of choice. Throughout the Android 6.0 *Marshmallow* previews, it was constantly evolving and sometimes breaking compatibility with older versions. [54] [6]

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 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. Rooting 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. Details and references of OS vulnerabilities can be accessed on pages like Common Vulnerabilities and Exposures or similar [32] [33]. These details are relevant for those creating rooting instructions or tools, but not for their users.
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 [81]. Rooting is usually bundled with installing a program called *Super User*. It is used to manage 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 device is then nonfunctional and no software cannot be executed anymore, even the OS itself. [56]
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 the secure folder on the device. The user has no rights to access the APK in this folder and thus cannot copy it, unless they have rooted the device. 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.

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. Now a license verification is done by a central authority, i.e. an application store. Google, as the main of contributor to Android and provider of its biggest store, started this by providing the License Verification Library (LVL) as will be described in subsection 2.3.1.
Android's 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, both offer a license verification solution as well.

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 an application store, they are also offering different services to bind to their ecosystem.
There are different Chinese and niche stores as well. Niche stores with few users and offerings are less attractive subjects to attack. Some do not even have any license verification.
The stores in focus for this thesis have to fight piracy in order make their store attractive for developers.

The scope of this thesis are Google, Amazon and Samsung since they have a critical mass in users and a license verification mechanism.

2.3.1 Google's License Verification Library

In order to tackle the copy protection problematic and to give the developer community a possibility to fight piracy, Google introduced the LVL on the 07/27/2010 [28]. It is easy to use and free of charge. The documentation can be found on the Android developers website [14].
Google's approach is based on a network service (see figure 2.7). It allows to query the trusted Google Play license server in order to determine whether the user has a valid license.

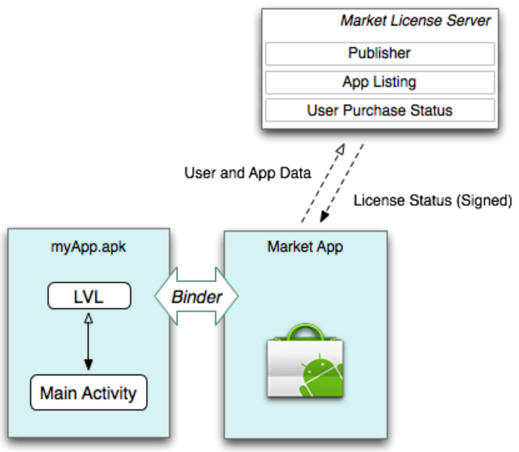


Figure 2.7: Google's implementation of license checking [14]

The source code for the LVL is provided by Google inside the Android SDK. It has to 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. Creating a unique implementation by changing its code, structure and control flow makes it more difficult target.
The LVL can be integrated in only a few steps. It is called and based on the result, the normal processing continues or the application terminates. The logic of the application itself has not to be altered.
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 and does license verification there. When an application is registered in Google Developer Console and the entry is created, an application specific public/private key pair is generated. The use is explained later on. [16]
After an application 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. [49] [14]
The first extension is the licensing permission in the *AndroidManifest.xml* (see code snippet 2.1). It is necessary for the LVL to work. When the application tries to contact the server for license verification and the permission is not set, an exception is thrown. [16] [9]

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

Code Snippet 2.1: Permission to check the license in *AndroidManifest.xml* [9]

The second extension is the implementation for the asynchronous handling of the callback the license verification call when the process is completed. 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. [14] [16] [9] [49]

```java
private class MyLicenseCheckerCallback implements LicenseCheckerCallback {

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

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

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

Code Snippet 2.2: LVL callback

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

        mChecker.checkAccess(mLicenseCheckerCallback);
```

Code Snippet 2.3: Setting up the LVL

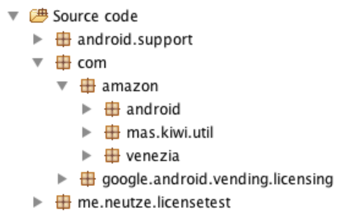
The third extension is the license verification call which can be seen in code snippet 2.3. The *LicenseChecker*, is initiated by passing three arguments. The first argument is the application which is provided by Android. The second argument is the public encryption key. It is retrieved from the Google Developer console and has to be stored in the code by the developer. The third argument is the policy. The policy decides what happens with the response data, e.g. if it should be cached or requested every time. The developer has to define the policy. Google provides two example policies as part of the LVL. Policies require obfuscation to prevent root users from manipulating or reusing the license response data. An example obfuscator is included in the LVL. It is generated using a salt, the package name and the *ANDROID_ID*. The ID is created randomly when the user sets up the device for 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. [14] [16] [9] [49]
Upon execution, the information is passed to the Google Play Service client residing on the device. The 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 sent 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. [14] [16] [9] [49]
The LVL mechanism replaces the old copy protection with a new approach. Instead of preventing the distribution of the application, the LVL prevents the execution of the application without a license.
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. 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 chosen policy, internet connection is needed. [14] [16] [9] [49]

2.3.2 Amazon DRM

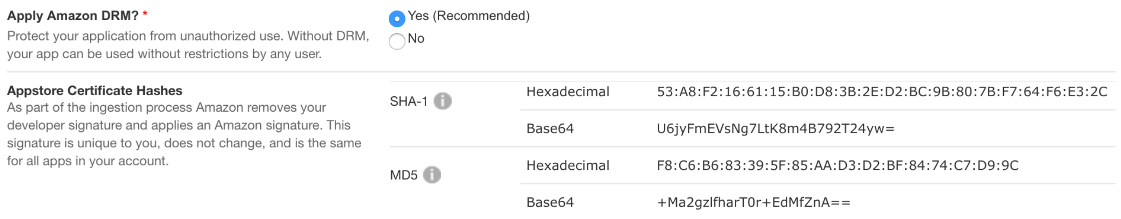
Amazon started its own application store in October 2010 [3] as an alternative to Google Play. The Amazon appstore opened to the public on the 03/22/2011 [4]. It can be used on all Android devices and *Fire* tablets. The store comes with its own Digital Rights Management (DRM) since the Google LVL only works with the Google Play Store. The DRM is called *Kiwi* as it can be seen in the reverse engineered code in figure 2.8.
The prerequisites for using *Kiwi* are similar to the ones of the LVL, since the developer requires a developer account on the Amazon Developer Service platform. According to the description, library is aimed to "Protect your application from unauthorized use. Without DRM, your application can be used without restrictions by any user."[2].
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 2.9). 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 key itself is available on the developer platform as seen in figure 2.9. [2]



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

Figure 2.8: Amazon DRM structure of a decompiled application



Apply Amazon DRM? *

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

[✓ Yes (Recommended)] No

Appstore Certificate Hashes

As part of the ingestion process Amazon removes your developer signature and applies an Amazon signature. This signature is unique to you, does not change, and is the same for all apps in your account.

SHA-1	Hexadecimal	53:A8:F2:16:61:15:B0:D8:3B:2E:D2:BC:9B:80:7B:F7:64:F6:E3:2C
	Base64	U6jyFmEVsNg7LtK8m4B792T24yw=
MD5	Hexadecimal	F8:C6:B6:83:39:5F:85:AA:D3:D2:BF:84:74:C7:D9:9C
	Base64	+Ma2gzlfharT0r+EdMfZnA==

Figure 2.9: Developer preferences in the Amazon developer console [2]

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

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

Code Snippet 2.4: Amazon DRM injection in the *onCreate()*

The license verification requires the Amazon's Appstore application to be installed on the device. Similar to the Google Play server, the store application has the knowledge of whether the user has purchased the application and thus is allowed to use it or not. As soon as the user logs in once into the Appstore application, the information is downloaded to the device. 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 online connection to the Amazon server, *Kiwi* is depending on an internet connection 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 application is not owned by the current user, respectively that the Amazon Appstore is required and cannot be found.

2.3.3 Samsung DRM

Another major player in the Android business is Samsung [29]. With *SamsungApps*, renamed to *GalaxyApps* in July 2015, they offer an application store to their Android devices. Application distributed in that store can be protected using *Zirconia* [65]. *Zirconia* is Samsung's implementation of a license verification library, using a server client model.
The way the library works is similar to the LVL. In order to prevent unauthorized usage of the application, the library queries the Samsung server to verify the license of the use. The library can be downloaded from Samsung in an archive file [65]. It contains the compiled *Zirconia* library as a .jar and additional native libraries for the different processor architectures. 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 which means three parts have to be added.
First, the required permissions have to be added to the *AndroidManifest.xml*. *Zirconia* needs access to the internet and the device information. The implementation can be seen in code snippet 2.5.
The second addition is the implementation of the *LicenseCheckListener* in code snippet 2.6. It has methods for the two possible results, valid or invalid license verification. 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 in code snippet 2.7. *Zirconia* handles everything on its own. The developer just has to call the *checkLicense()* method.

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

Code Snippet 2.5: Permission in the *AndroidManifest.xml* [65]

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

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

Code Snippet 2.6: *Zirconia* callback

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

Code Snippet 2.7: Setting up the Zirconia

Zirconia always follows the same internal pattern when the license check is executed. First, it is queried for a stored response from previous verifications. If a stored response exists and is valid, the check passes and no internet connection is required. Otherwise *Zirconia* sends information to the server to verify the license. This includes information about the device and the application. The server evaluates whether the user is authorized to use the application and replies accordingly. This is stored on the device. [65]

2.3.4 Summary

All license verification libraries are working inside the application and query a trusted source. For the LVL and *Zirconia*, the trusted source is a server while for *Kiwi* it is the store application. They do not prevent redistribution or copying, but enforce the authorization when the application is run. The result of the verification is always binary, from the view of successfully run the application, it is even unary. Only one result is acceptable, which is license verified. In the further analysis it will be shown that this unary mechanism is an easy target for attacks, such as executed by Lucky Patcher.
The functionality of the libraries and in fact any test can be abstracted as a simple *yes/no* check as seen in figure 2.10.

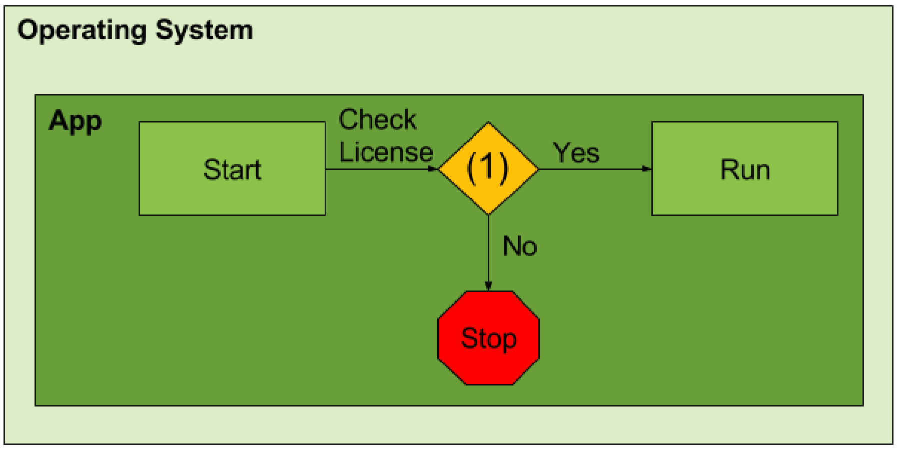


Figure 2.10: Abstraction of the current license verification mechanism. The library is represented by *(1)*.

2.4 Code Analysis

For the approach chosen, the code needs to be reverse engineered on different abstraction layers. First, it is described how to extract APK from the device. Then the extraction of the *classes.dex*, the conversion to smali and the reverse engineering of the Java code are shown. Finally, the use of diff is explained.

2.4.1 APK Extraction

Code analysis is performed on a desktop computer using various tools. The APK has to be extracted from the device and transferred onto the computer, because the tools cannot analyse the application while it is still on the device. 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 when pulling 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, the location of the application must be known to the user. The package manager can be used to 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 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 sent to the computer.[1]

2.4.2 Abstraction Levels

Code analysis is performed on three different abstraction levels.
The first level is the original dex bytecode contained in the *classes.dex*. As will be shown chapter 3, it is target of the attack.
The second level is the smali code. It is used to represent the dex bytecode in a readable way and to identify the result of the changes.
The third level is the Java code. It is the best presentation to interpret and understand the changes in the context of methods or classes.
The Java code can be decompiled from the dex bytecode since the build process can be reversed as seen in figure 2.11.

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 the places

[1]*Root* is neither a prerequisite to access the APKs nor to alter their *classes.dex* file

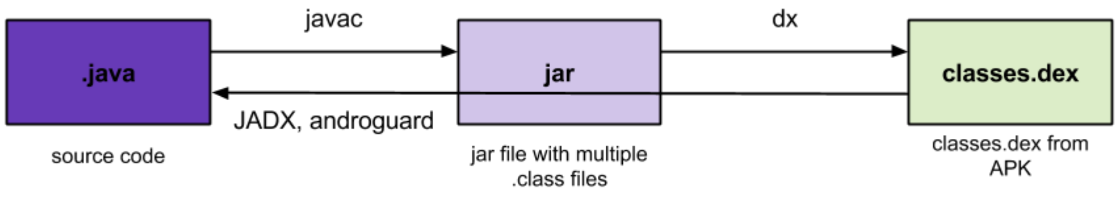


Figure 2.11: Bidirectional relation between Java .class and .dex bytecode [54]

in the bytecode that 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* 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 [60]. 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

Smali Analysis

The smali code is disassembled from dex bytecode using *baksmali* [45]. 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[45]. The syntax is loosely based on the Jasmin syntax [45]. Smali takes the APK and disassembles its *classes.dex* file. The output is a 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.

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

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

An example of smali code can be seen in code snippet 2.11. It is easier to understand than 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 workflow 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: Example of smali code

Java Analysis

Java code suits best for the analysis of the new behavior since its presentation is close to the what people are programming. It has names variable and method which makes it better understandable. The representation is close to how the developer implemented the application.
As seen in figure 2.11, the .dex files can be reengineered to Java .class files. The reengineered Java code maintains the semantics, but does not exactly match the source code. As we will see, code attacked may contain instructions that are removed by the decompiler. Sometimes a decompiler does not deliver an acceptable result, that is why DAD and JADX are used alternatively.
DAD, is a short name of "DAD is A Decompiler". It is part of Androguard [5], a reverse engineering tool for Android. It works with the dex bytecode and does not require third party tools like dex2jar [61]. Code snippet 2.12 shows how it is used in the command line interface to decompile the APK into Java code.

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

Code Snippet 2.12: Script to decompile to Java using androguard

JADX [68] is the second decompiler. The decompilation is directly done from the APK's dex bytecode to Java code and can be performed in the command line interface as seen in Code snippet 2.13.

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

Code Snippet 2.13: Script to decompile to Java using JADX

2.4.3 Pinpointing the Attack

The amount of code on each abstraction levels is quite substantial. In a code altering attack, most of the code stays the same, i.e. changes are only punctually. Diff, a standard command line tool, is used to compare two sources with little differences. By comparing the different code abstractions of the original APK with the cracked application, changes done by Lucky Patcher are identified. 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.

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

Code Snippet 2.14: Script to compare the original and manipulated APK using diff

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 2.15.

```
13 00 00 01 33 R0 ?? ?? 54 ?? ?? ?? 71 10 ?? ?? ?? ?? 0C 01
13 W0 00 01 33 00 00 01 54 ?? ?? ?? 71 10 ?? ?? ?? ?? 0C 01
```

Code Snippet 2.15: Example of diff for dex code

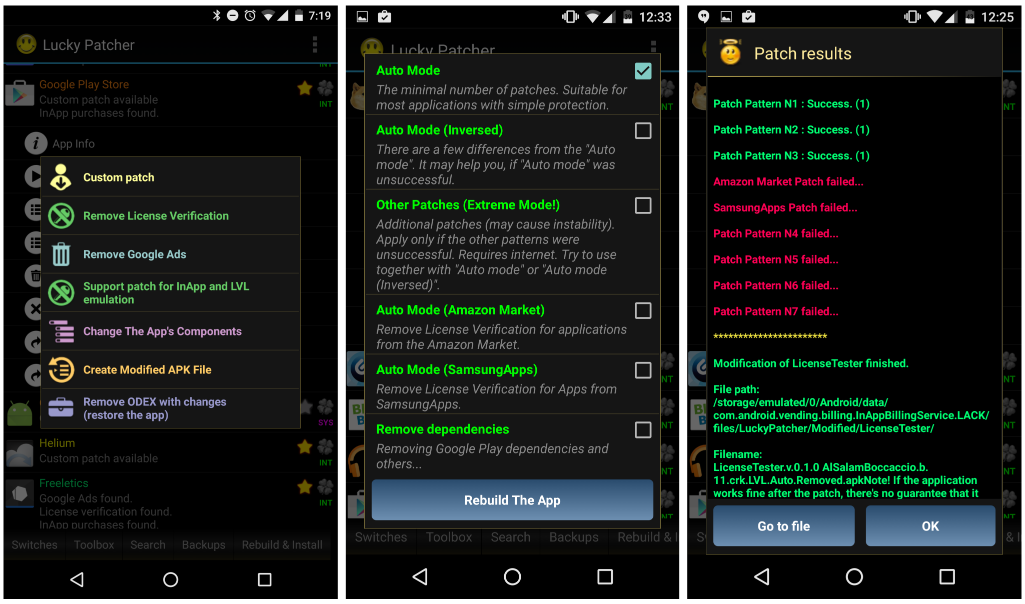
3 Analysis of Lucky Patcher

Android is under attack. Piracy is quite common and of great concern.
This thesis focuses on one of the most popular cracking applications, Lucky Patcher, and its attack strategy.

3.1 Lucky Patcher

On its 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"[27]. It is written by a developer calling himself ChelpuS and currently on version is 6.0.7 (03/03/2016).
Lucky Patcher offers different options to crack and alter applications. It is not guaranteed that any of the options will succeed. [27]
Lucky Patcher requires no technical knowledge and offers automatic cracking to non professionals. This combination makes it a popular and an effective tool with a high potential for creating damage.

The application can be downloaded as an APK from the official website [27] and installed on any device, *root* is not neccessarily required. 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. Only when all features of Lucky Patcher are needed, the device has to be rooted [27].
When an application is selected, a submenu offers various actions, e.g. to get information about the application 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. The custom patch option applies changes specifically designed for a chosen application.
Universal patches can be applied for the removal of the license verification libraries and Google Ads or the rebuilding of the application to use the emulated LVL and in-app billing API.
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



Auto Mode [✓]
The minimal number of patches. Suitable for most applications with simple protection.

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

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

Auto Mode (Amazon Market) ☐
Remove License Verification for applications from the Amazon Market.

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

Remove dependencies ☐
Removing Google Play dependencies and others...

Rebuild The App

Patch results

```
Patch Pattern N1 : Success. (1)

Patch Pattern N2 : Success. (1)

Patch Pattern N3 : Success. (1)

Amazon Market Patch failed...

SamsungApps Patch failed...

Patch Pattern N4 failed...

Patch Pattern N5 failed...

Patch Pattern N6 failed...

Patch Pattern N7 failed...

************************

Modification of LicenseTester finished.

File path:
/storage/emulated/0/Android/data/
com.android.vending.billing.InAppBillingService.LACK/
files/LuckyPatcher/Modified/LicenseTester/

Filename:
LicenseTester.v.0.1.0 AlSalamBoccaccio.b.
11.crk.LVL.Auto.Removed.apkNote! If the application
works fine after the patch, there's no guarantee that it
```

Go to file　OK

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

approach is the creation of a modified APK. 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 installed on the device after removing the original APK or transferred to another device.
Both approaches offer the custom and *universal* patches.

The focus in this thesis is on the *universal* patches for removing of the license verification by creating a modified APK.
There are five *universal* modes and two *universal* patches available for removing the license verification 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.
The patching starts when a mode is selected. When the process is finished, a result screen is shown as seen in figure 3.1 on the right. It indicates that different patches were applied.

3.2 Selecting the Approach

When analysing software applications, a white box or black box approach can be taken. The white box approach looks inside the software and uses knowledge obtained from

the code. The black box approach makes no assumptions about the content of the software, but only observes inputs and outputs of the application.
First general pros and cons of each approach are listed, then they are evaluated for the scenario of this work and finally an approach is chosen.

3.2.1 White Box vs Black Box

In order to prepare a decision on which approach to take, the following pros and cons are considered for each of the two.

	White Box	Black Box
Pro	• ability to find edge cases • the deeper the understanding of the code, the better the understanding of attack vectors	• strong when dealing with large code base • clear view on effects allows abstract view on mechanisms
Con	• requires intimate knowledge of the code • requires full understanding of the algorithms	• potentially difficult to define proper test cases • edge cases might not be triggered

Table 3.1: Pros and cons of white and black box analysis

3.2.2 Evaluation

There is no source code available for Lucky Patcher. For this reason, the code has to be reconstructed from the *classes.dex* of the APK. After testing several decompilers, DAD and JADX are both selected to decompile the dex bytecode to Java code, in order to get the best possible starting point to gain an understanding. Most notable are the following two points:

- some blocks fail to successfully decompile to Java
- flat structure and unusually large classes, e.g. *listAppsFragment.java* has over 17.000 lines of code

While decompiled code is always obscure, there are indicators that the developer has purposely taken steps to obfuscate the meaning of the code of Lucky Patcher. Examples are the launcher activity residing in a package looking like provided by Google or the flat hierarchy without clear architectural patterns.
In order to get familiar with the functionality of Lucky Patcher, an application containing license verification is cracked. Lucky Patcher reports the use of seven named *Patch Patterns* and two *Patches*. Trying to identify the changes to the application done by Lucky Patcher, the dex bytecode of before and after patching is compared. Most notable are the following two points:

- confined and understandable changes between original and cracked *classes.dex* are identified
- presence of five patching modes suggest a manageable amount of test cases

3.2.3 Selection

The goal of the analysis is the deep understanding of the mechanisms applied by Lucky Patcher. In order to propose countermeasures, an abstract understanding of these mechanisms has to be achieved. The chosen approach needs to go deep enough and reveal the essence of what Lucky Patcher does.
While the look into the white box approach was discouraging, the black box immediately revealed remarkably confined changes traceable to a purpose. When continuing with the black box analysis, the changes are starting to fall into recurring patterns.
The conclusion is to choose the black box test. Additional insight into the code is sought where appropriate.

3.3 Blackbox Analysis

One of the difficulties of black box analysis, is the definition of the right test cases. As a reminder, the scope of the analysis in this thesis are the five *universal* modes and two *universal* patches for removing the license verification and creating a modified APK. The five modes and two patches form the first dimension in the variations of test cases. Different applications will trigger different reactions in the black box and form a second dimension.
Initially, fourteen applications are selected for the analysis, but quickly it becomes apparent that only a limited number of reactions is triggered. For every test case run, Lucky Patcher announces success and failure of each of nine applied patterns and patches.

Lucky Patcher's different modes and patches are described in figure 3.1 on the right.

- 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[1])* - "Removes License Verification for Apps from SamsungApps"

The *Extreme Mode!* can either be applied on its own or combined with the the *Auto Mode* or the *Auto Mode (Inversed)*. This accounts for the five *universal* modes for the LVL, while the patches for Amazon and Samsung are in addition.
A reference application with an implementation according to the tutorial of LVL [9], is created for the black box test. This application is called *LicenseTest*. It gives full control and knowledge, as the source code is available and allows an analysis of Lucky Patcher's attacks in the most comprehensive way.
When testing Amazon and Samsung, *LicenseTest* is uploaded to the stores, including the LVL. Of course, the LVL is disabled because it is not supported on those platforms. For Samsung, *Zirconia* is implemented according to subsection 2.3.3 while Amazon injects *Kiwi* as discussed in subsection 2.3.2.
Among the other applications are Runtastic Pro[64], version 6.3, and Teamspeak 3[75], version 3.0.20.2, for the LVL and A Better Camera [1], version 3.35, for the Amazon DRM. These apps were chosen since they were already owned by the author and, upon request, the developers approved them to be mentioned in this thesis. The rest of the applications is not mentioned here, since the approval was not able to be obtained.
While initially the test was organized by executing each application in each mode, it becomes immediately apparent that each mode triggers specific patterns and patches and they drive the behavior. As patterns can be triggered intentionally in a test case, the whole test is finally organized to analyse one pattern at a time. This will be described in the next section.

[1]SamsungApps is called GalaxyApps, see subsection 2.3.3

3.4 Patch Patterns and Patches

When executing test cases, Lucky Patcher offers a full report of what it calls *Patch Patterns* and *Patches* and their result. While Lucky Patcher provides seven different patterns for the LVL, called *Patch Pattern N1..N7*, it gives to Amazon and Samsung one patch each and calls them *Amazon Market Patch* and *SamsungApps Patch*, respectively.
When patching the LVL, Lucky Patcher's choice of patterns is dependent on the selected mode. Table 3.2 shows the patching modes and the *Patch Patterns* or *Patches* applied by Lucky Patcher.

	Patch Patterns									
Mode	N1	N2	N3	N3i	N4	N5	N6	N7	A	S
Auto	X	X	X		X					
Auto (Inversed)	X	X		X	X					
Extreme						X	X	X		
Auto+Extreme	X	X	X		X	X	X	X		
Auto (Inversed)+Extreme	X	X		X	X	X	X	X		
Amazon		X							X	
Samsung		X								X

Table 3.2: Overview of *Patch Patterns*/*Patches* applied by each mode

In order to identify the changes done by each individual *Patch Pattern* and *Patch*, the code of the original and cracked APK are compared using diff. The changes are inspected on dex, smali and Java level with the tools explained in section 2.4.
When the dex code of an application is modified, the checksum and the signature of the file have to be recalculated which can be seen in the diff. In addition, the whole APK has to be signed again, as described in subsection 2.2.1. This does not change the application logic, which is focus of the following.

As shown below, the changes observed are only replacements of bytecode and very limited in scope and number. They do not add or remove bytecode, i.e. they do not add or remove logic blocks. Instead Lucky Patcher enforces a certain control flow, by forcing an evaluation to be *true* or *false* as needed and ignoring actual results of method calls. This is done by manipulating a single instruction, either opcode or argument. The surrounding context remains untouched.
All this is done on the dex bytecode level, which opens the question on how the target of the change is located. As will be shown, Lucky Patcher uses bytecode search patterns to locate where the change should be placed and bytecode replace pattern to execute it. The search pattern consists of the target instruction to be manipulated and the context,

while the replace pattern has the same context with the target instruction changed.
A look at the Lucky Patcher sources shows that it contains strings, which are formatted like bytecode, and include the target instructions and the context in a masked form. These strings are located in the Java class *com/chelpus/root/utils/odexrunpatch.java* of Lucky Patcher and appear in the context of the *Patch Pattern* they belong to. Each *Patch Pattern* can have multiple search and replace bytecode pattern pairs, implementing the same change in different contexts, which are tried when the *Patch Pattern* is used.
The context in a search pattern is given as a mask of a fixed length with fixpoints given as explicit bytecode tuples and placeholders. When Lucky Patcher tries to position a search pattern, it searches for a sequence of bytecodes matching the fix points and the target instruction as given by the mask. An example is shown in code snippet 3.1. On success, it substitutes that sequence with the replace pattern.

```
@@ Search pattern @@
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28

@@ Replace pattern @@
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 12 ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28
```

Code Snippet 3.1: Bytecode patterns - Target instructions are green (search) and red (replace), fixpoints are blue and placeholders are given as *??*

Before explaining the patterns in detail, additional information has to be provided. In the .dex file analysis, a simplified presentation of the binary data as *0a* instead of hexadecimal values like *0x0a* is chosen for improved readability in the diff files. When converting .dex files to smali files, arguments like x in dex code are shown as vx in smali and constants stay the same. The opcode defines which of its arguments are constants or variables

3.4.1 License Verification Library

Lucky Patcher identifies seven *Patch Patterns* for the LVL, *N1..N7*, which are analysed in the following.

Patch Pattern N1

Patch Pattern N1 is present in all patching modes, except the solo extreme mode. It targets the *verify()* method of the *LicenseValidator* class in the *com/google/android/vending/licensing/* folder. This method is responsible for decrypting and verifying the response from the license server [15].
The *Patch Pattern* swaps *1a* and *0f* in their order, as seen in the dex code in code snippet 3.2.
As seen the following, all other *Patch Patterns* make use of search and replace patterns as described in section 3.4. *Patch Pattern N1* is the only one which does not make use of this mechanism. Instead a logic inside the code is used to determine the position and apply the change. As this is a one off and the focus is on the result, this is not described in detail.

```
@@ Patch Result 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 3.2: Diff on dex level for *Patch Pattern N1*

When looking at the smali code, the two switched bytecode tuples can be identified as blocks of a switch statement. The swap of switch cases *0x1* und *0x2* can be seen in the diff of code snippet 3.3.

```
@@ Patch Result N1 @@
- 0x1 -> :sswitch_e0
- 0x2 -> :sswitch_d5
+ 0x1 -> :sswitch_d5
+ 0x2 -> :sswitch_e0
```

Code Snippet 3.3: Diff on Smali level for *Patch Pattern N1*

In the Java code snippet 3.4, 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 treated as valid.

```
@@ Patch Result 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 3.4: Diff on Java level for *Patch Pattern N1*

The result is the voiding of the *verify()* switch case. It handles the user as verified even though the response code is *NOT_LICENSED* at the cost of the response code *LICENSED_OLD_KEY*. This response code is sent when the developer changes the signature of the license verification process and updates the application. This happens rarely but when it does, a newer version of the application has to be cracked.

Patch Pattern N2

Like *Patch Pattern N1*, *Patch Pattern N2* is applied in all patching modes, except the solo extreme mode. It targets the *LicenseValidator* class's *verify()* method.
The changes in the .dex file can be seen in code snippet 3.5. First, the mask of the search pattern has to match with the fixpoints on the dex bytecode. When a matching is successful, the replace pattern is applied. The result of *Patching Pattern N2* is the replacing of the instruction *0a 05* with the instruction *12 15*. *Patch Pattern N2* has one pair of search and replace pattern in the code of Lucky Patcher. Lucky Patcher uses the variable *S1* for the second bytecode tuple. The target of the opcode altered remains the same, while the source argument is modified. The placeholder indicates the source is set to *1*.

```
@@ Patch Result N2 @@
- 0c 05 6e 20 9d 4a 53 00 0a 05 39 05 2d 00 1a 05
+ 0c 05 6e 20 9d 4a 53 00 12 15 39 05 2d 00 1a 05

@@ Bytecode Pattern N2 @@
0A ?? 39 ?? ?? 00
12 S1 39 ?? ?? 00
```

Code Snippet 3.5: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N2*

The smali in code snippet 3.6 shows the altered instruction. Instead of moving the result of the preceding function to variable *v5*, *v5* is set to *true*.

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

Code Snippet 3.6: Diff on Smali level for *Patch Pattern N2*

The result in the Java code (see code snippet 3.7) shows 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.

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

Code Snippet 3.7: Diff on Java level for *Patch Pattern N2*

The consequence is that *verify()* of the signature is still executed but the result is ignored. The program flow continues after *verify()* as if the signature was valid.

Patch Pattern N3

Patch Pattern N3 has two variants. While is applied *N3* is applied in the *Auto Mode*, *N3i* is applied in the *Auto Mode (inversed)*. It targets the *allowAccess()* method inside the *APKExpansionPolicy* and *ServerManagedPolicy*. These two classes are the policy examples provided by Google and located *com/google/android/vending/licensing/* folder [15].
Both *Patch Patterns* have an opposing result. One bets on *true* being a positive result to ensure execution of the application, while the other one bets on *false*. While *Patch Pattern N3* is replacing the *01* with *11*, *Patch Pattern N3i* does the opposite by replacing *11* with *01* (see code snippet 3.8 and code snippet 3.9).
The source code of Lucky Patcher contains three categories of search and replace bytecode patterns for *Patch Pattern N3*. The first category are the four bytecode patterns for *Patch Pattern N3*. They replace the target instruction's arguments with the the variable *S1*, i.e. the source arguments of the target instructions are set to *1*. This category is betting on *true* to continue.
The second category are the four bytecode patterns for *Patch Pattern N3i*. Instead of setting the sources to *1*, the bytecode patterns replace the two target instructions' sources with *0*, as the placeholder *S0* indicates. This category is betting on *false* to continue.

```
@@ Patch Result N3 @@
- 12 10 12 01 71 00 a6 89 00 00 0b 02 52 84 c1 1c
+ 12 10 12 11 71 00 a6 89 00 00 0b 02 52 84 c1 1c
# 13 05 00 01 33 54 09 00 53 84 bc 1c 31 02 02 04

@@ Bytecode Pattern N3 @@
12 ?? 12 ?? 71 ?? ?? ?? ?? ?? 0B ?? ?? ?? ?? ?? ?? ?? ?? ?? 33 ?? ?? ??
   53
12 S1 12 S1 71 ?? ?? ?? ?? ?? 0B ?? ?? ?? ?? ?? ?? ?? ?? ?? 33 ?? ?? ??
   53
```

Code Snippet 3.8: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N3*

```
@@ Patch Result N3i @@
- 12 10 12 01 71 00 a6 89 00 00 0b 02 52 84 c1 1c
+ 12 00 12 01 71 00 a6 89 00 00 0b 02 52 84 c1 1c
# 13 05 00 01 33 54 09 00 53 84 bc 1c 31 02 02 04

@@ Bytecode Pattern N3i @@
12 ?? 12 ?? 71 ?? ?? ?? ?? ?? 0B ?? ?? ?? ?? ?? ?? ?? ?? ?? 33 ?? ?? ??
   53
12 S0 12 S0 71 ?? ?? ?? ?? ?? 0B ?? ?? ?? ?? ?? ?? ?? ?? ?? 33 ?? ?? ??
   53
```

Code Snippet 3.9: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N3i*

```
@@ Bytecode Pattern N3x @@
13 00 00 01 33 R0 ?? ?? 54 ?? ?? ?? 71 10 ?? ?? ?? ?? 0C 01
13 W0 00 01 33 00 00 01 54 ?? ?? ?? 71 10 ?? ?? ?? ?? 0C 01

70 ?? ?? ?? ?? ?? 6E ?? ?? ?? ?? ?? 0C ?? 38 ?? ?? ?? 62 ?? ?? ?? 6E ?? ??
    ?? ?? ?? 0A ?? 44 00 01 00 2B 00 ?? ?? ?? ?? 62 ?? ?? ?? 11
70 ?? ?? ?? ?? ?? 6E ?? ?? ?? ?? ?? 0C ?? 38 ?? ?? ?? 62 ?? ?? ?? 6E ?? ??
    ?? ?? ?? 0A ?? 12 10 00 00 2B 00 ?? ?? ?? ?? 62 ?? ?? ?? 11
```

Code Snippet 3.10: Search and replace bytecode pattern to apply for *Patch Pattern N3x*

While the four patterns of *N3* and *N3i* have the same context but different guesses, the two patterns of *N3x* are different in their structure but enforce the same logic. The first bytecode pattern of *N3x* targets implementations using *13*, an opcode to move a 16 bit constant instead of a 4 bit constant. The second bytecode pattern of *N3x* is applied when the value is retrieved from an array (opcode *44*). The original instruction is replaced by a move of a 4 bit constant *12 10*.

When looking at the smali diff in code snippet 3.8, the dex code is translated to the initialization of *v1*. While N3 sets *v1* to 1, N3i sets *v1* to *0*.

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

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

Code Snippet 3.11: Diff on Smali level for *Patch Pattern N3* and *N3i*

```
@@ Patch Result N3 @@
- result = false;
+ result = true;
# ...
# return result;

@@ Patch Result N3i @@
- result = true;
+ result = false;
# ...
# return result;
```

Code Snippet 3.12: Diff on Java level for *Patch Pattern N3* and *N3i*

The resulting Java code is shown in code snippet 3.12. *Patch Pattern N3* voids the result of the *allowAccess()* method by already initializing the result value with the desired outcome. While N3 is targeted towards code where the default return value is *false*, *Patch Pattern N3i* is used when the default value is *true*.
Both *Patch Patterns* attack the class's *allowAccess()* method which evaluates whether the verification result is according to the policy or not. Both return variables are

initialized with the same value. This makes the result independent of the outcome of the verification. As LP cannot predict if the answer *true* or *false* allows continuation of the license verification process, it offers both versions and the user has to try which is the right one. This is the reason for the *Auto Mode(inversed)*.

Patch Pattern N4

Patch Pattern N4 is part of the auto and auto inverse patching modes. The target of the *Patch Pattern* is the *LicenseChecker* class of the LVL, which is responsible for initiating the license check in its *checkAccess()* method [15].
As seen in code snippet 3.13, it replaces *38* with *33*. In addition, the one bytecode pattern found in Lucky Patcher's source code reveals that the arguments of *33* are set to *00* instead of the original bytecode tupel.

```
@@ Patch Result N4 @@
- d5 70 00 00 0a 00 38 00 0e 00 1a 00 5a 20 1a 01
+ d5 70 00 00 0a 00 33 00 0e 00 1a 00 5a 20 1a 01
# 6c 29 71 20 74 34 10 00 72 10 c6 70 08 00 1e 07

@@ Bytecode Pattern N4 @@
0a ?? 38 ?? 0e 00 1a ?? ?? ?? 1A ?? ?? ?? 71 ?? ?? ?? ?? ?? 72
0a ?? 33 00 ?? ?? 1a ?? ?? ?? 1A ?? ?? ?? 71 ?? ?? ?? ?? ?? 72
```

Code Snippet 3.13: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N4*

In the smali code snippet 3.14 this change can be identified as replacing *if-eqz* with *if-ne*. The opcode *if-eqz* takes one argument *v0* and implicitly fills the remainder of the tuple with *0*, while *if-ne* takes two arguments and interprets the implicit *0* as *v0*.

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

Code Snippet 3.14: Diff on Smali level for *Patch Pattern N4*

The Java code in code snippet 3.15 shows the result of the change. In the original code, the result of *mPolicy.allow()* was checked. In case the policy did not allow to continue, i.e. the result was *false*, the condition block was executed. The change results in the check for inequality of the result of *mPolicy.allow()* to itself. Since the result of the method is identical if called twice, the condition is never fulfilled and the condition block is never called.

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

Code Snippet 3.15: Diff on Java level for *Patch Pattern N4*

Patch Pattern N4 ensures that the result of textitcheckAccess() is never considered.

Patch Pattern N5

As part of the extreme mode, *Patch Pattern N5* targets the *LicenseValidator*'s *verify()* method. It targets implementations with other context than the basic implementation of the LVL.
The diff of the dex code in code snippet 3.16 shows the replacing of *0a* with 12. *Patch Pattern N5* makes use of three bytecode patterns with different context, found in the source code of Lucky Patcher. The location of the change is dependent the eight fixpoints of each search pattern mask.

```
@@ Patch Result N5 @@
# 28 7e 0b 00 0c 00 07 1b 07 01 28 c6 22 00 a5 10
# 70 10 da 70 00 00 5b 01 01 2c 12 01 46 01 02 01
- 71 10 ab 7d 01 00 0a 01 59 01 fb 2b 12 11 46 01
+ 71 10 ab 7d 01 00 12 01 59 01 fb 2b 12 11 46 01

@@ Bytecode Pattern N5 @@
22 ?? ?? ?? 70 ?? ?? ?? ?? ?? 5B ?? ?? ?? 12 ?? 46 ?? ?? ?? 71 ?? ?? ?? ??
    ?? 0A ?? ?? ?? ?? ?? 12 ??
22 ?? ?? ?? 70 ?? ?? ?? ?? ?? 5B ?? ?? ?? 12 ?? 46 ?? ?? ?? 71 ?? ?? ?? ??
    ?? 12 ?? ?? ?? ?? ?? 12 ??
```

Code Snippet 3.16: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N5*

The modification done by *Patching Pattern N5* is moving to a variable a constant (*const/4*) instead of a result (*move-result*).

```
@@ Patch Result N5 @@
- move-result v1
+ const/4 v1, 0x0
```

Code Snippet 3.17: Diff on Smali level for *Patch Pattern N5*

The result of patching can be seen in the Java diff in code snippet 3.18. The original code parses the response code from the data record of the server. After applying the patch, the response data is still parsed, but the result is ignored and the response code is set to *LICENSED*.

```
@@ Patch Result N5 @@
- if (data.responseCode != responseCode) {
+ data.responseCode;
+ if (LICENSED != responseCode) {
```

Code Snippet 3.18: Diff on Java level for *Patch Pattern N5*

Patch Pattern N6

Patch Pattern N6 is part of the extreme mode and, similar to the *Patch Pattern N1*, *N2* and *N5*, it attacks the *verify()* method in the LVL's *LicenseValidator* class.
This *Patch Pattern* has only one bytecode pattern pair, which targets three bytecode tuples of the .dex file which can be seen in code snippet 3.19. The first changes value is *38* to *12*. The second value is *06* which is replaced by *00*. The third change is the replacing of *4a* by *00*.

```
@@ Patch Result N6 @@
- 38 0a 06 00 32 4a 04 00 33 5a 21 01 1a 00 ab 15
+ 12 0a 00 00 32 00 04 00 33 5a 21 01 1a 00 ab 15
# 71 10 ed 21 00 00 0c 00 6e 20 ee 21 90 00 6e 10

@@ Bytecode Pattern N6 @@
38 ?? 06 00 32 ?? 04 00 33 ?? ?? ?? 1A ?? ?? ?? 71
12 ?? 00 00 32 00 04 00 33 ?? ?? ?? ?? ?? ?? ?? ??
```

Code Snippet 3.19: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N6*

The change of the code structure is more visible in the code snippet 3.20. The first results in the initialization of *p2* with *0*. The second change is required to maintain the length of the sequence. The opcode *const/4* has two bytecode tuples as argument, while the original *if-eqz* has four - the opcode tuple, the argument tuple and a target consisting of two tuples. The difference is fixed by changing the third and fourth bytecode tuple to a *nop* operation (*00 00*). It is presented 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.

```
@@ Patch Result 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 3.20: Diff on Smali level for *Patch Pattern N6*

The changes on Java level of the *Patch Pattern* are presented in code snippet 3.21. In the original code the if statement tests whether the response code is not one of the desired values. After patching, the evaluation is always *false* and the code block inside is never used.

```
@@ Patch Result 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 3.21: Diff on Java level for *Patch Pattern N6*

This *Patch Pattern* prevents the execution for cases where the *verify()* has to handle response codes that are neither *LICENSED*, *NOT_LICENSED* or *LICENSED_OLD_KEY*. Instead the method proceeds as if the response code is valid.

Patch Pattern N7

The final *Patch Pattern* for the LVL is *Patch Pattern N7*. Inside the lvl, it patches the *ILicenseResultListener* class's *onTransact()* method, which receives the asynchronous response from the license server [15]. In addition to the lvl classes, the *Patch Pattern* is applied to all classes residing in the `com/android/` package.
Patch Pattern N7 has seven bytecode pattern pairs, which all replace *0a* with *12*, but in different contexts.

```
@@ Patch Result N7 @@
# 00 00 00 00 2e 00 00 00 12 10 2c 05 23 00 00 00
# 6f 58 3e 03 54 76 0a 00 0f 00 1a 01 cb 1d 6e 20
# 94 03 17 00 28 fa 1a 01 cb 1d 6e 20 83 03 16 00
- 6e 10 87 03 06 00 0a 01 6e 10 8a 03 06 00 0c 02
+ 6e 10 87 03 06 00 12 01 6e 10 8a 03 06 00 0c 02
# 6e 10 8a 03 06 00 0c 03 6e 40 1c 14 14 32 28 e5

@@ Bytecode Pattern N7 @@
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 12 ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28
```

Code Snippet 3.22: Diff on dex level and search and replace bytecode pattern to apply for the *Patch Pattern N7*

In smali, this shows to be the replacement of moving a result into *v1* by initializing *v1* to *0*.

```
@@ Patch Result N7 @@
- move-result v1
+ const/4 v1, 0x0
```

Code Snippet 3.23: Diff on Smali level for *Patch Pattern N7*

Similar to *Patch Pattern N2*, *Patch Pattern N7* attacks by initializing a variable with *false* instead of moving a result of a method into it.
Lucky Patcher uses the search pattern for this *Patch Pattern* on a broader scope and it is applied when the context matches. This can result in undesured changes in other classes and thus lead to instability.

```
@@ Patch Result N7 @@
- this.verifyLicense(data.responseCode, data.signedData, data.signature);
+ data.responseCode;
+ this.verifyLicense(LICENSED, data.signedData, data.signature);
```

Code Snippet 3.24: Diff on Java level for *Patch Pattern N7*

This *Patch Pattern* ensures the *verifyLicense()* method is always executed with the constant *LICENSED* instead of the result of the licensing check.

3.4.2 Amazon Market Patch

Kiwi library is injected by Amazon and cannot be customized by the developer. This means only one *Patch* is necessary. The *Amazon Market Patch* has two bytecode patterns, each is applied once when patching. The first class targeted 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 Market Patch* has two bytecode pattern to replace *38* with *33*.

```
@@ Patch Result Amazon @@
- 1d 01 6e 20 c4 2e 21 00 0c 00 38 00 05 00 12 10
+ 1d 01 6e 20 c4 2e 21 00 0c 00 33 00 05 00 12 10
# 1e 01 0f 00 12 00 28 fd 0d 00 1e 01 27 00 00 00

@@ Bytecode Pattern Amazon @@
6E 20 ?? ?? ?? ?? 0C 00 38 00 05 00 12 10 ?? ?? 0F 00 12 00 ?? ?? 0D 00 ??
     ?? 27 00
6E 20 ?? ?? ?? ?? 0C 00 33 00 ?? ?? 12 10 ?? ?? 0F 00 12 00 ?? ?? 0D 00 ??
     ?? 27 00
```

Code Snippet 3.25: Diff on dex level and search and replace bytecode pattern to apply for the *Amazon Market Patch*

The patch replaces *if-eqz* with *if-ne* as seen in code snippet 3.26. The opcode *if-eqz* evaluates only the one argument, comparing it to zero. *if-ne* takes two arguments, comparing them for non-equality. Having both arguments the same always yields *false* and the condition code block is never executed.

```
@@ Patch Result Amazon @@
- if-eqz v0, :cond_1f
+ if-ne v0, v0, :cond_1f
```

Code Snippet 3.26: Diff on Smali level for the *Amazon Market Patch*

The Java code presentation helps to interpret the changes of the attack. In the class *b.java*, the if block, formerly repsonsible for reponse codes not *APPLICATION_LICENSE*, is now never called as the condition is always *false*.
The method modified in the d.java class tests whether a string is not *null*. After the patching, the functions returns always *true*.

```
@@ Patch Result Amazon @@
- if( ! v0.equals("APPLICATION_LICENSE")) {...}
+ if(v0.equals("APPLICATION_LICENSE") != v0.equals("APPLICATION_LICENSE"))
    {...}
```

Code Snippet 3.27: Diff on Java level for the *Amazon Market Patch*

The changes void the if statement for checking whether the response code is *APPLICATION_LICENSE*. The result is forced to be always *true* and thus the license verification always passes.

3.4.3 SamsungApps Patch

Similar to the Amazon *Kiwi* library, Samsung's *Zirconia* library cannot be modified by the developer. For this reason, cracking the library requires only one *Patch*, consisting of the two *Patch Pattern S1* and *S2*. The *SamsungApps Patch* is applied on the *LicenseRetriever* and *Zirconia* class in the *com/samsung/zirconia* package. *S1* is applied on both classes once and *Patch Pattern S2* is applied twice but only on the *Zirconia* class.
S1 uses six bytecode pattern pairs while *S2* has two bytecode pattern pairs. While the bytecode patterns *S1* replace *d6* with *00*, the ones for *S2* use *12* instead of *0a*.

```
@@ Patch Result S1 @@
# 13 08 09 00 6e 30 b9 4a 85 0c 0c 08 71 10 6d 4a
- 08 00 0c 08 6e 10 66 4a 08 00 0a 06 32 d6 0a 00
+ 08 00 0c 08 6e 10 66 4a 08 00 0a 06 32 00 0a 00
# 59 e6 fd 20 22 08 9f 08 70 30 c1 49 e8 0b 27 08

@@ Bytecode Pattern S1 @@
13 ?? 09 00 6E ?? ?? ?? ?? ?? 0C ?? 71 ?? ?? ?? ?? ?? 0C ?? 6E ?? ?? ?? ??
     ?? 0A ?? 32 ?? ?? ?? 59 ?? ?? ?? 22 ?? ?? ?? 70 ?? ?? ?? ?? ?? 27
13 ?? 09 00 6E ?? ?? ?? ?? ?? 0C ?? 71 ?? ?? ?? ?? ?? 0C ?? 6E ?? ?? ?? ??
     ?? 0A ?? 32 00 ?? ?? 59 ?? ?? ?? 22 ?? ?? ?? 70 ?? ?? ?? ?? ?? 27
```

Code Snippet 3.28: Diff on dex level and search and replace bytecode pattern to apply for the *Patch Pattern S1*

```
@@ Patch Result S2 @@
# a6 dc 1e 00 17 00 00 00 54 30 07 21 71 10 ea 49
# 00 00 0c 00 54 31 07 21 71 10 e5 49 01 00 0c 01
# 54 32 07 21 71 10 e6 49 02 00 0c 02 71 30 ce 49
- 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

@@ Bytecode Pattern S2 @@
54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ?? 54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ??
    54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ?? 71 ?? ?? ?? ?? ?? 0A ?? 0F ?? 00
    00
54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ?? 54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ??
    54 ?? ?? ?? 71 ?? ?? ?? ?? ?? 0C ?? 71 ?? ?? ?? ?? ?? 12 S1 0F ?? 00
    00
```

Code Snippet 3.29: Diff on dex level and search and replace bytecode pattern to apply for the *Patch Pattern S2*

The result of *Patching Pattern S1* is that the *if-eq* statement instead of comparing two variables only compares *v0* to itself. The result of this comparison is always *true*. *Patching Pattern S2* has the effect that *v0* does not return the result of the previous method but always *true*.

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

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

Code Snippet 3.30: Diff on Smali level for the *SamsungApps Patch*

The presentation in code snippet 3.31 contains the changes in Java code. Instead of continuing execution based on the result of *foo()* (*LicenseRetriever*'s *receiveResponse()*), it executes *foo()*, but ignores the result and proceeds as if it was *true*. In the method *checkerThreadWorker()* of the *Zirconia* class, *Patching Pattern S1* voids the check of the response code and always continues as if the response code was valid.
Patching 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*.

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

@@ Patch Result S2 @@
- result = foo();
+ result = true;
# ...
# return result;
```

Code Snippet 3.31: Diff on Java level for the *SamsungApps Patch*

The result of applying the patch is that the license file checks are voided and return the verification as *true*. In addition, response 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

As described in section 3.3, different applications are tested by attacking them with Lucky Patcher. Lucky Patcher does not guarantee to be successful with one or more patching mechanisms. Amazon and Samsung patches are always successful, since the library attacked is not changeable by the developer.

All modifications done by Lucky Patcher change a condition using a license verification of some kind and determining to continue or not, into a condition using a constant determining to continue in any case. In other words, it changes a binary decision, using the result of a verification, into a unary decision to just go ahead. Despite the attack, the individual license verifications are still executed, but the results ignored. In all of this, the complexity and level of security of the license verification itself is not relevant. An abstract presentation of the unary decision mechanism is presented in figure 3.2.

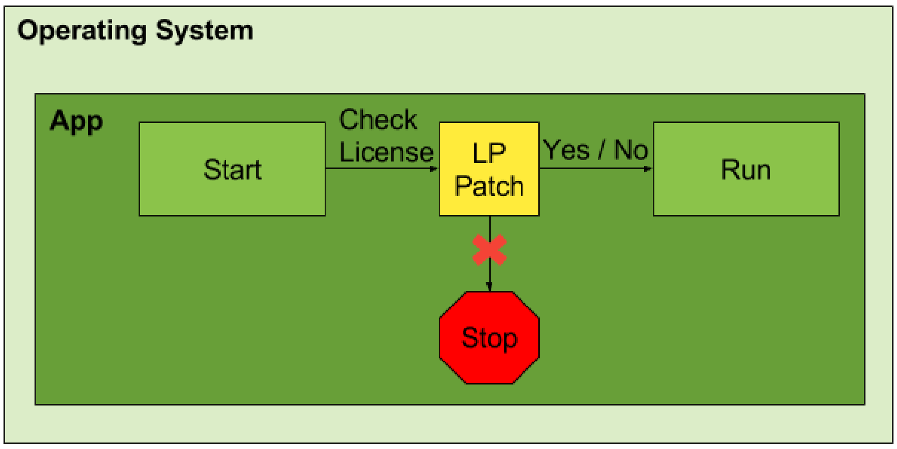


Figure 3.2: Attack on the license verification mechanism by Lucky Patcher

4 Countermeasures for Developers

Now that that the methodology of Lucky Patcher's attacks is analysed, suggestions to prevent it are proposed.
The first section covers methods to improve the current implementation of the LVL, an implementation of the LVL using native code and suggests additional tampering checks and obfuscation.
The second section proposes a content driven approach, replacing verifications reducible to unary decisions, with the provision or withholding of complex content.

4.1 Fortification

This section proposes countermeasures building on the license verification mechanism. The first suggestion aims to make the LVL implementation unique as suggested by Google. This changes the bytecode and may render some of the search patterns of Lucky Patcher useless.
The second suggestion is to get rid of the dex bytecode targeted by the automatic patching of Lucky Patcher.
The third suggestion introduces an additional layer of checks.
Finally, obfuscation is discussed to prevent future reverse engineering efforts.

4.1.1 Unique Implementation

Attacking with Lucky Patcher is often successful because developers do not customize the LVL at all. One reason for not customizing the LVL is that protection against casual piracy feels sufficient for the developer. Casual piracy is the copying of APKs from one device to another.
Another reason is that the developers do not know how to fortify the library and thus they do not want to spend additional effort [47].
While the idea in general is valid for all license verification libraries, its implementation is only possible for the LVL. It requires access to the source code, which is available for the LVL, as opposed to Samsung's and Amazon's libraries.
Lucky Patcher's attack is reliant on successfully applying the patching patterns. For this reason, actively avoiding these patterns should always be the first step to challenge

Lucky Patcher.
This approach is only possible when using the LVL, as it requires access to the source code. Modifying and improving the LVL does not only protect from patterns used by Lucky Patcher, increasing complexity of the application's bytecode makes it unique and harder to reengineer. In fact, modifying the lvl is what google strongly suggests. [47]
There are three areas the developer should focus on when modifying the LVL [47].

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

The core logic's 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 much as possible, while retaining the original function. Isomorphic code changes can include:

- replacing a switch statement with a set of if statements and adding additional code between them (see pattern N1)
- deriving new values for response codes and checking for these values in the further proceeding (see pattern N6)
- removing unused code for interfaces, e.g. implementing the policy verification inline in the *LicenseValidator* instead of over an interface (see patterns N2, N4, N5 and N6)
- moving the LVL package functionality into the application package (see patterns N2 and N7)
- implementing functions inline where possible (see patterns N2, N5 and N7)
- making actions in the decompiled code difficult to trace by moving routines to unrelated code (counter intuitive from traditional software engineering)
- implementing radical response handling, e.g. killing the application without a hint as soon as an invalid server response is detected (results in bad user experience)

These are only examples and creativity is welcome, since the resulting implementation should be unique. [47]

The entry and exit points can be attacked by creating a counterfeit version of the LVL that implements the same interface. A 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. [47]
The attackers do not only target the core of the LVL, but also the handling of verification results. This can be prevented by handling the mechanism in a separate activity. This saves 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. [47]
These modifications are easy to apply and make the implementation unique. There is an unlimited number of ways to implement them. It needs to be stated, that each is easily reengineered and patched, but this makes it hard to attack automatically with an *universal* solution. A determined attacker, willing to invest time and work in disassembling and reassembling, can eventually find a weakness in the code. This is the usual race of arms between making things secure and finding new ways to break them. The knowledge gained can be used to create a custom patch for Lucky Patcher, 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. [47]

4.1.2 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 in C or C++ and compiled to machine code native to that platform (native code).
Usually the NDK targets CPU intensive tasks, such as game engines and signal processing, but it can be used for any other purpose as well. Google suggests using it only if necessary, since it increases the complexity of an application [10]. This is a desired side effect when implementing the LVL natively. Native code, opposed to 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 their 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. [57]
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 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. A single 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 attacked.

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. The developer has to consider the trade offs between security and simplicity.

4.1.3 Tampering Protection

When circumventing the LVL the code has to be modified. There are different indicators for actual and likely modifications on the application. Indicators for actual modifications are forced debuggability, installation from a source different than the store or the changed signature of the application. When *root* is available or Lucky Patcher is installed on a device, these are indicators for likely breaches in security.
These indicators can be tested and the program can be interrupted when in doubt. For detected actual modifications, no reason should be given for the killing of the application, as any reason given can be used for reengineering purposes. Indicators of an unsafe environment should trigger interruption and a dialog, explaining the risk. Of course interruption of the application results in a bad user experience. The tests for these indicators are implemented as seen in figure 4.1.

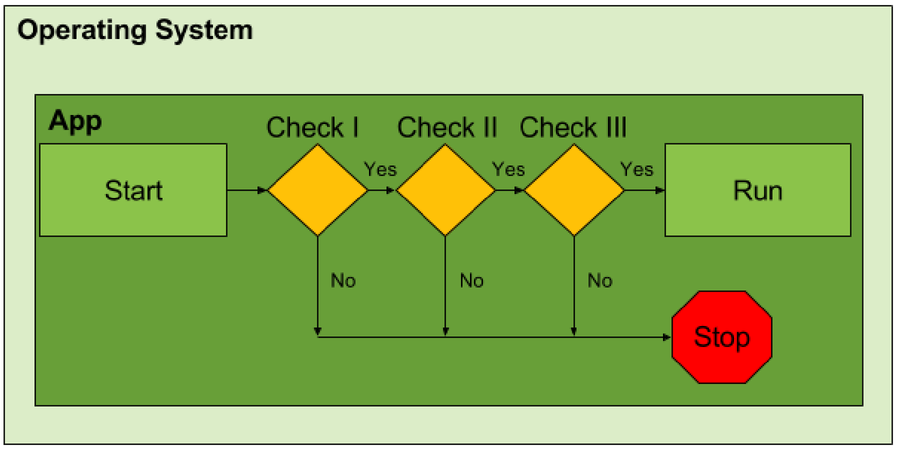


Figure 4.1: Additional checks as additonal layer of security

Note that all these tampering countermeasures have the binary decision making, similar to the license checks process. They can be circumvented by a reengineering effort and turning them into the unary scenario.
On reengineering, the code has to be analysed in order to find, understand and patch them. The countermeasures should 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. Obfuscation, which should be applied as a standard, makes reengineering more difficult.
None of this prevents Lucky Patcher from working, but it offers an additional layer of security.

Debuggability

Enabling debugging allows the developer to use additional features for analysing the application at runtime, like printing out logs [13]. These features are used to gain information about the flow of the application and to reengineer functionality. With the results of this analysis, weak points are identified and custom patches are developed. The debug flag indicates that additional information for debugging can be provided. It is not set in release builds on the application stores, but it can be activated by changing it in the *classes.dex*. In order to prevent attackers taking advantage, the developer can check whether this flag is activated and the application is tampered. Code snippet 4.1

```
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

is an example for an implementation of this check. The debug flag can be acquired 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 these devices from running the application. The developer needs to communicate to the users the reasons for this strict policy, since there are a lot of users who use *root* for other reasons than cracking applications. Code snippet 4.2 is an example implementation.
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 is terminated.
Google has introduced an API providing similar checks, called SafetyNet [11]. It is used to check the "health and safety of an Android"[43]. It is said to be used in security critical applications like Android Pay and the reason for excluding rooted devices from the service [11] [51] [63].

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

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

        return result;
    }
```

Code Snippet 4.2: Example code for checking for *root*

Lucky Patcher

Having Lucky Patcher installed, is a strong indicator that the user is pirating applications. The check can be extended to detect additional unwanted applications by adding their package name to the check [82]. 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 acquire information whether the Lucky Patcher package is installed. In case information is available and thus the application is installed, the check stops the application. Again, it might be useful to communicate up front that the application will not run, if Lucky Patcher is present.

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

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

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

        } catch (final PackageManager.NameNotFoundException ignored) {
        }

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

        return result;
    }
}
```

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

Sideload

As modified APKs can be created and installed on the target device or any other, checking for *root* and Lucky Patcher is 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* [30]. 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. The application will not work when retrieved from a non listed store.

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

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

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

        return result;
    }
```

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

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 working by accident"[47].

Signature

When modifying an application, it has to be signed once more as described in subsection 2.2.3. Since Lucky Patcher does not have the developer's key, it has to sign the application using a different key.
When storing the original key's hash in the code, it can be compared to the current one. In case they differ, execution can be interrupted.

```
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

The code for the signature check can be seen in code snippet 4.5. The original key's hash has to be provided (see line 53). The application signing key's hash is fetched from the package information (line 57ff). In case it cannot be retrieved or does not match, the check terminates the application.
The approach is similar to Google Maps inside an application. When launching the map, the application sends the SHA1 signature and the API key to the server, which verifies whether the application is allowed to display the map [42].

4.1.4 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. Android applications are at high risk of being reverse engineered, as its bytecode is easily decompiled. In order to make reverse engineering difficult and delay the development of custom patches for Lucky Patcher, obfuscation is introduced. Obfuscation is an easy to apply protection and should be used in every application. An

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* [53]. Some dex obfuscators exist as well, like *DexProtector* [34].
Basic obfuscators do not protect against automated attacks since they do not alter the bytecode. They can be applied to the standard version of the LVL, but it is no protection since the source code is known. The full potential of obfuscation is unleashed, when combined with a unique implementation that has to be reengineered in order to be understood. It makes the attackers' work much more time consuming, up to the point where the effort is no longer profitable. When the attacker does not find proper class and method naming, it is harder to identify the purpose of the particular part analysed. It makes it much harder to develop a custom patch. [47]
Some methods cannot be obfuscated, as the Android framework relies on calling them by name, e.g. *onCreate()*. The developer should avoid implementing license verification related code inside these unobfuscated methods, since attackers will look into these methods. [47]

Applying obfuscators does not directly protect from Lucky Patcher attacks. When the implementation of the LVL is unique and obfuscated, 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.

4.2 Content Driven Application

Content driven applications are introduced to replace the Achilles' heel of license verification culminating in a binary decision, with mechanisms where the result cannot be guessed and injected. The content of the application is dependent on the status of the application and can usually not be guessed.
Content cannot be guessed, but it might get stolen. Encryption is introduced to prevent that.

4.2.1 Content Server

This approach enforces users to log into a server and not relying on the verification response itself, but the returned content for the application, only accessible on successful verification. This content cannot be guessed by Lucky Patcher. A presentation of this approach is in figure 4.2.

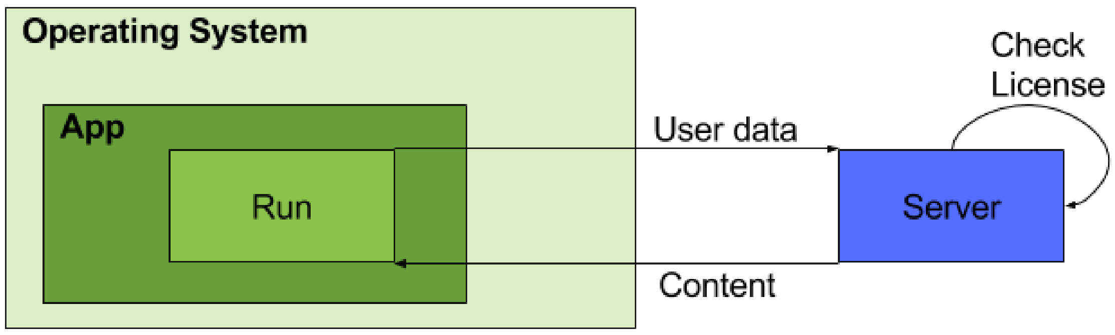


Figure 4.2: Abstraction of an application and a content server

An implementation can be illustrated by looking at the application Spotify [69] as an example. Instead of verifying the license locally on the device, the user has to enter their credentials and send them to the server. In case the credentials are valid, the user has the right to receive content. The content, the music in this case, is no longer on the device 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, this content is not available inside the application until the user is authorized on the server. Thus attacks on the application itself do not yield the desired result anymore.

In general, a content server is a solution against piracy, but it has downsides as well. The first problem is that this architecture cannot be applied to all applications. If there is no complex enough content that can be moved to the server, it doesn't work.
The second problem is the more complex overall architecture. Instead of using Google's solution for handling the verification and only implementing the application logic on the device, a full size client server structure for the content has to be developed and implemented.
The third problem are the additional resources needed. When outsourcing parts of the application on a server, money is needed for the server.
The fourth problem is the requirement for a permanent online connection. It limits the freedom of users and creates additional internet traffic and costs. This is not accepted

by all users.

The content server mechanism protects the developers IP when the core algorithm is moved to the server. This prevents attackers not only from using the application for free, but also from reconstructing the core functionality and implementing it somewhere else.

4.2.2 Content Encryption

Encryption is introduced as part of content based countermeasure. The content driven approach relies on content being provided by a server. This can be done unencrypted under the condition the user is verified, or encrypted where only verified users get the proper key. This key has to be protected against theft.

Encryption

Encryption can be applied on different levels inside the application. This thesis introduces three different ones.

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.
If application critical strings are encrypted, like server addresses, the application is unable to work without decryption. If the strings are output strings or pictures, the application will work, but the user will not understand the output, because it is still encrypted and has no meaning to the user.
Figure 4.3 shows the abstract implementation of resource decryption.

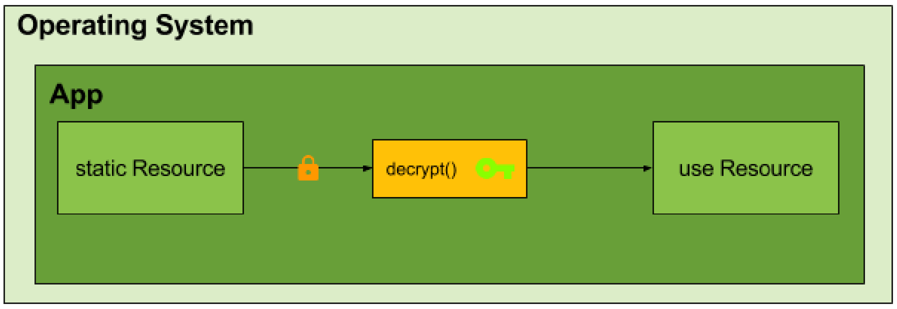


Figure 4.3: Encrypted resources have to be decrypted before they are used or displayed

There is a possibility to steal decrypted the static resources from authorized applications and insert them unencrypted into a recreated APK. This is a significant effort for a specific application.

Encryption - Action Obfuscator

The second approach is to use encryption as a form of obfuscation. The idea is to have a single method (*abstractMethod()*) to delegate all other method calls according to an encrypted parameter.
When an attacker does a static analysis of the code, the link between the initial call and target method is not apparent. It forces the attacker to use a dynamic analysis method instead. The mechanism can be fortified even more, when encrypted arguments are passed.
An abstract presentation of the mechanism can be seen in figure 4.4.

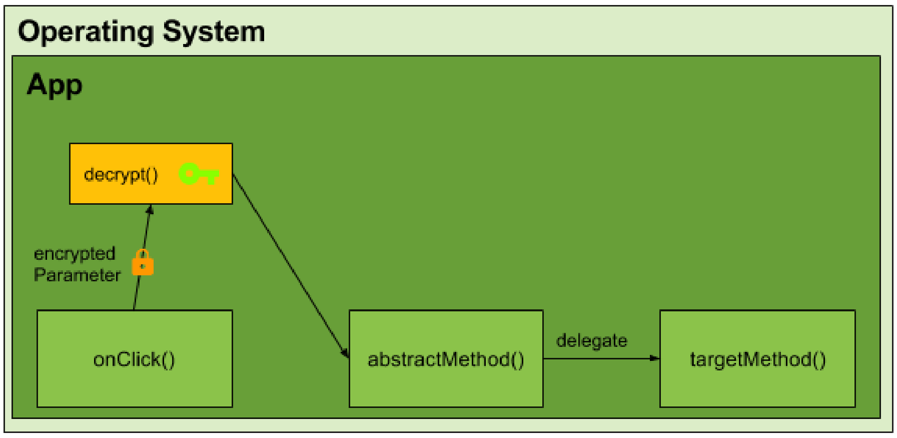


Figure 4.4: Encrypted paramters to obfuscate method dependencies

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.

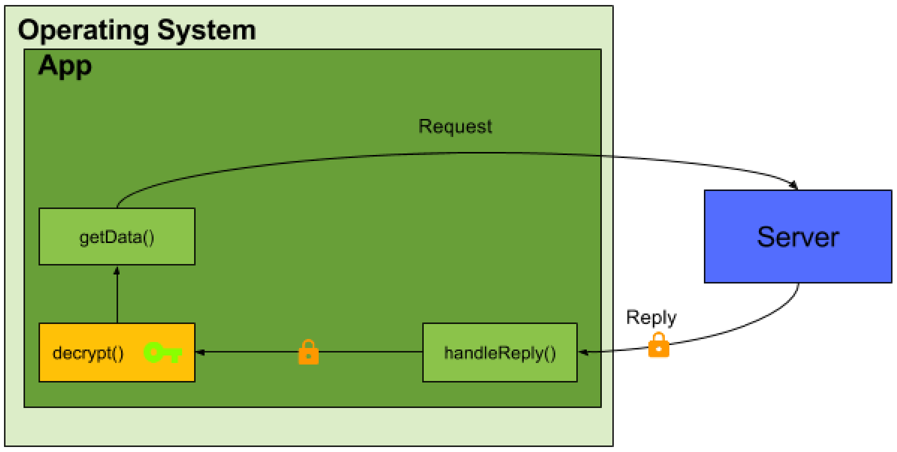


Figure 4.5: Encrypted communication with a server

Key Storage

In order to use encrypted content, a decryption key is needed.

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 (see figure 4.6). This works similar to the license verification, but affects the whole content.
The key can either be retrieved from the server for each decryption action or it can be cached on the device. Caching should be favored, since getting the key for each action not only requires permanent online connection, but slows down the application and generates additional traffic.
In the caching scenario, on *decrypt()* call, the application tries to retrieve a cached cryptographic key, which it may have received and stored earlier. 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 and when the check is successful, the decryption key is provided.

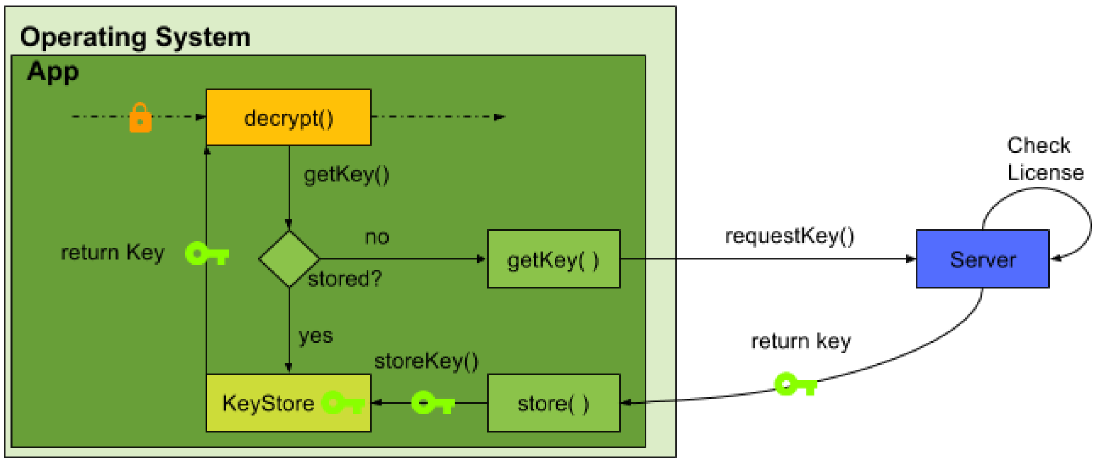


Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

A cached key can be stolen [76] and the encryption cracked this way. A key needs to be protected against theft and should be changed periodically, e.g. when updating the version of the application. When running a content server, the key can be user or device specific and changed more often.

Key Storage - Secure Element

A sophisticated way to provide and store a cryptographic key, is the use of a Secure Element (SE).
A SE is a tamper-resistant platform which can be used to securely host simple applications and cryptographic keys [39]. There are different form factors for SEs, i.e. Universal Integrated Circuit Card (UICC), embedded SE and microSD [39]. For Android, the microSD is 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 Android device to support USB On-The-Go (OTG) [79]. The SE is accessed over reads and writes to its filesystem. Since the SE has to be small to fit the size of a microSD card and is powered by the host system, its hardware capabilities are constrained. The result is a performance as low as 25MHz as compared to the GHz of the system's CPU. This does not allow complex computations on the SE [70]. For this reason, the usage of the SE is restricted to simple tasks, like decryption. The decryption key is either provided up front on the SE or the SE implements a secure mechanism to get the key. In any case, it needs to relate to the key used for encryption. The management of this relationship can be quite complex, e.g. involving a certification authority. 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 for a SE implementing a decryption method can be seen in figure 4.7.

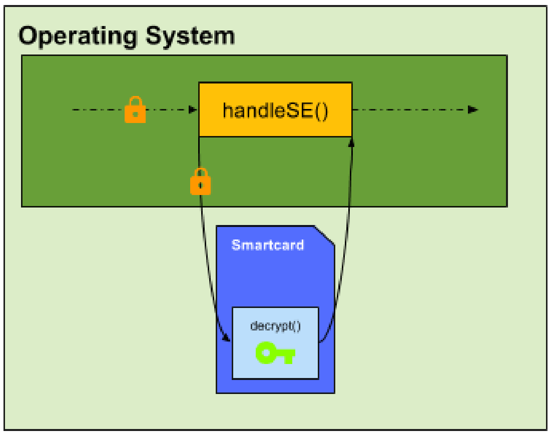


Figure 4.7: Decryption by using a smartcard

At the moment, there are some problems with the integration of SEs.

- it invovles extra hardware
- not all devices have a spare microSD card slot or support OTG
- no standardized communication protocols for the SE across manufacturers

The first problem is that this requires extra hardware and the hardware has to be around to use the functionality.
The second problem is the SE mounted on an external interface for devices without an microSD card slot. It is no convenient solution and an easy target of an attack.
The third problem is that some devices without microSD card do not support OTG. For example, the Nexus 7 (2012) and the Nexus 6P neither have the capability to use a microSD card. While the Nexus 7 (2012) is supposed to have OTG, it does not work with the tested SE, while the Nexus 6P does not support OTG at all.
The fourth problem is the lack of standards. An application written to use SE would need to have variants for each device type and manufacturer. The SD Association has proposed a standard, if adopted the usage of SE might increase [66].

A similar approach is the Trusted Execution Environment (TEE). The TEE offers a secure virtualized instance on the CPU of a device. The environment is isolated from the rest of the device and used to protect sensitive data and processes. Since it is on the main processor, it has more computational power than a SE and requires no additional hardware. There is already an Android implementation called *Trusty* [8]. Android's DRM framework is already making use of the benefits The TEE can be used to decrypt the protected content in a location where the key cannot be stolen. At the moment, there is no support for third party applications since the applications have to be packaged with the *Trusty* kernel, signed and verified by the bootloader. [8] [40]

5 Conclusion

This thesis analyses Lucky Patcher's attacks on license verification libraries using a black box approach and isolating attack vectors. Based on this, more or less disruptive countermeasures are proposed.

5.1 Summary

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

The first chapter starts with the introduction of software licensing, its goals and the reason it needs to be enforced. The current situation and problems with licensing on Android is portrayed. Different research in this field is presented as related work.

The second chapter explains the fundamentals needed to understand why software piracy is a problem on Android. Android and the steps needed to run an application are explained. Then the license verification libraries, which are target of Lucky Patcher, are introduced. Finally the tools used for the code analysis are portrayed.

The third chapter is all about the Android cracking application Lucky Patcher. First the functionality is presented. Then the selection process for the analysis approach is discussed and black box analysis explained. The findings of this black box analysis are presented in the following and abstracted.

The fourth chapter suggests two different types of countermeasures. The first part is about improvements to the current state of the license verification libraries. The second part introduces a content driven approach more disruptive to the initial architecture of an application.

5.2 Discussion

The chosen black box approach is effective, focusing the view on the one vulnerability attacked - single verification decisions that can be voided to be unary.

There are several levels of the race of arms between developers protecting and pirates stealing IP.
On the lowest level, attack vectors of a specific search pattern can be blocked, but Lucky Patcher will implement an answer and the general problem of the unary vulnerability is not fixed.
On another level, dex bytecode is always reengineerable, opening a plethora of attack vectors. This vulnerability can be fixed by replacing bytecode with native code, but native code will be attacked in different ways.
Even higher concepts of protection will be cracked as part of the race of arms, e.g. encryption, SEs or TEEs.

Security is always part of a trade off against openness. For instance, the prevention of tampering APKs with self-signed signatures could be improved by introducing a certification authority. Thinking this further would lead to a *walled garden*, contradicting Android's philosophy of openness.

5.3 Future Work

There are several developments going on that will change the security landscape in the Android world.
Every version of Android closes several vulnerabilities but also opens new ones. Sometimes, the whole architecture is updated, e.g. the introduction of ART. With regards to Lucky Patcher, ART has the ability to do away with dex bytecode entirely, but in the real world, compatibility needs to be observed and non ART APKs need to be supported for the foreseeable future. This means APKs cannot omit .dex files and remain vulnerable.
At the moment, SE are not standardized and *universal* support not guaranteed by a reasonable number of devices. SEs open whole a range of possibilities which can only be tapped, when SEs reach a critical level of availability.
As seen in subsection 2.2.1, APKs are signed and protected on a certain level by self-signed certificates. Between this level and a walled garden, e.g. total control by Google, there has to be a viable trade off, making use of certification and certification authorities.

List of Figures

List of Tables

List of Code Snippets

Bibliography

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

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

[3] 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).

[4] 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).

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

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

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

[8] Android. *Trusty TEE*. URL: https://source.android.com/security/trusty/index.html#third-party_trusty_applications (visited on 03/11/2016).

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

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

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

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

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

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

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

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

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

[18] 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).

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

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

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

[22] 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.

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

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

[25] 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).

[26] 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).

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

[28] 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).

[29] 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).

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

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

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

[33] 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).

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

[35] 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).

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

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

[38] C. Fung. *Andriod IPC: Shared Memory with ashmem, MemoryFile and Binder*. URL: https://vec.io/posts/andriod-ipc-shared-memory-with-ashmem-memoryfile-and-binder (visited on 03/08/2016).

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

[40] GlobalPlatform. *GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide*. URL: http://www.globalplatform.org/mediaguidetee.asp (visited on 03/11/2016).

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

[42] 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).

[43] Google Developers. *SafetyNet*. URL: https://developers.google.com/android/reference/com/google/android/gms/safetynet/SafetyNet (visited on 03/09/2016).

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

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

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

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

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

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

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

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

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

[53] 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).

[54] J. Levin. *Dalvik and ART*. Dec. 2015. (Visited on 01/09/2016).

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

[56] 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).

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

[58] Nikolay Elenkov. *Code signing in Android's security model*. URL: http://nelenkov.blogspot.de/2013/05/code-signing-in-androids-security-model.html (visited on 03/12/2016).

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

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

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

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

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

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

[65] Samsung. *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).

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

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

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

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

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

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

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

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

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

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

[76] 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).

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

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

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

[80] 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).

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

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