DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Johannes Neutze, B. Sc.

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Analyse von Android Crackingtools und Untersuchung geeigneter Gegenmaßnahmen für Entwickler

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

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

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

Acknowledgments

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

Assumption

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

Abstract

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

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

- priacy problem
- android different approaches
- luckypatcher attacks
- does not change methods, but whether their outcome is included
- some patterns can be tricked with mopdification but reverse enineerint to easy
- freedom vs walled garden, only able to stop piracy when certificate is checked
- a lot to do on android

Contents

1 Introduction

1.1 Licensing

Software Licensing is the legally binding agreement between two parties regarding the purchase, installation and use of software according to its terms of use. It defines the rights of the licensor and the licensee. The goal is to protect the software creator's IP or other features and enable him to commercialize it as a product. It defines the boundaries of usage for the user and prevents him from illicit usage [**uncgLicensing**]. 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 [**szCopy**].

1.2 Motivation

Licensing is also present in Android. With a market share of almost 82.8% in Q2 of 2015 [**androidShare**] it is the most used mobile OS. According to Google, this market share translates to over 1.4 billion active devices in the last 30 days in September 2015 [**androidDevices**]. This giant number of Android devices is powered by Google Play [**googlePlay**]. 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 [**statistaAppStore**]. 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 [**wiwoValue**].
The growth comes with advantages. Some time ago developers only considered iOS as

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

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

1.3 Related Work

Lucky Patcher and license verification have already been topic of scientific work.
In his master's thesis **bernhardSecurity** [**bernhardSecurity**] **bernhardSecurity** takes a look at license verification an in-app billing attacks. He comes to the conclusion that the libraries need an overhaul since they are easy to circumvent and have not been updated for a long time. This shows the urgency for further investigation on this topic.
munteanLicense's master's thesis **munteanLicense** [**munteanLicense**] presents an analysis of techniques to crack Android's license verification and implements a new approach. He introduces multiple general strategies, such as obfuscation and dynamic code generation, to fortify code. In the end he uses the insights from the analysis to suggest countermeasures and their effects. A similar approach is chosen for this thesis.
Other scientific papers, like **Jang:2013:PAA:2480362.2480673** in their paper **Jang:2013:PAA:2480362.248067**

2 Foundation

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

2.1 Software Piracy

According to Apple, 11 billion Dollars are lost each year due to piracy [**applePiracy**]. Software piracy is defined as the unauthorized reproduction, distribution and selling of software [**applePiracy**]. 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 **??**. The developer loses direct revenue when his IP is stolen and redistributed by a pirate without the developer's involvement. In case the application is offered for free, users do not have to pay and no revenue is generated. It is even worse when the pirated application is sold in another app store. In this case the pirate gets the profit which should be the developer's.
Revenue is not only lost when the application can be downloaded for free. There is also follow up revenue effected, when an applications is changed. There are two main types of indirect revenue. /newline The first type is in-app purchases. They are a popular source of income for so called freemium or lite versions of applications. In case of the the freemium app, the download is for free and includes all features. The developer makes the money with in-app purchases like cosmetic interface changes or in-game currency. The lite version application is a little bit different. The download is free as well, but the application comes with a restricted feature set or limited time of use. In order to take advantage of the full feature set, the user can buy the pro-license via an in-app purchase.
Apps can include a mix or various degrees of theses types. Pirates can disable the transaction of the payments for in-app purchase thus no earnings are generated for the

developer while the user can access the content.
The second type of indirect revenue is generated by showing in-app advertisements. When this feature is implemented, advertisements are shown inside the application and the developer is paid by views and clicks on the advertisements. Earnings generated by an applications are assigned according to the included Ad Unit ID [**googleAdmob**]. When an application is pirated, this ID can be replaced by the pirate's ID. Future revenues generated by advertisements will not be assigned to the developer but to the pirate.
Beside monetary issues, additional problems arise when the application is moved to a

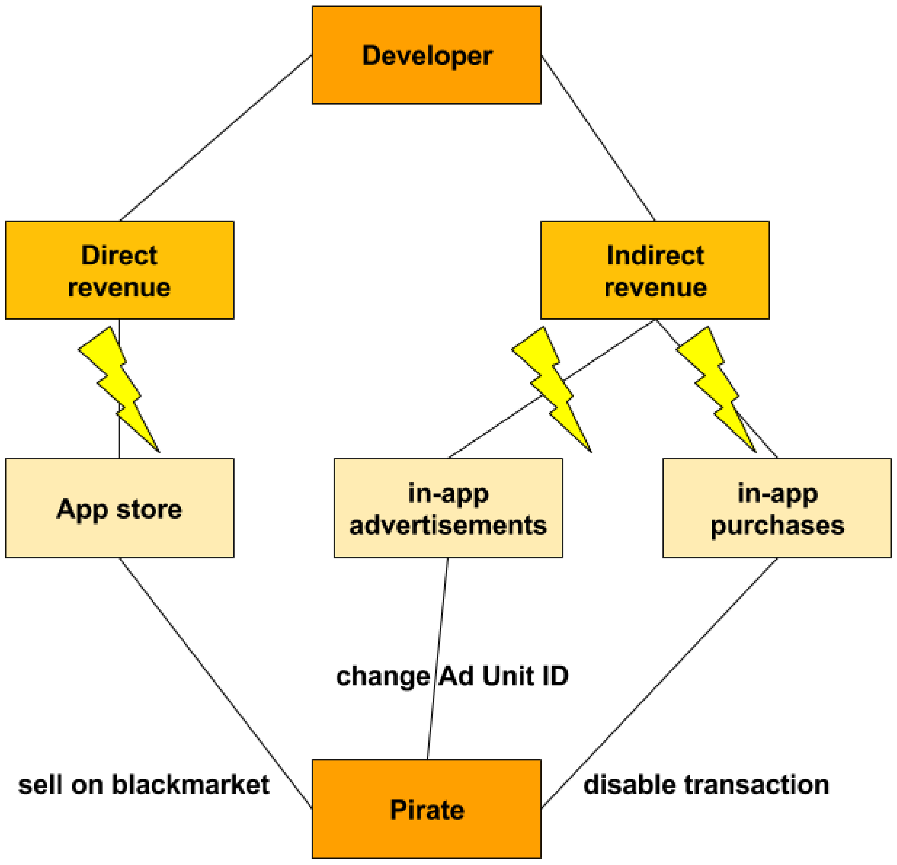


Figure 2.1: Different ways to generate revenue and how the pirate can cut them

black market store or website and distributed without the environment of an official app store. This results is the loss of control over the application for the developer. He can no longer provide support and updates for the application. The users will not get fixes for security issues and bugs. Users which do not know they are using a pirated

version will connect the unsatisfying behaviour to the developer. This results in the loss of reputation and revenues not even connected to this application.
If an application uses server resources, the developer can neither monitor the growth in the usage of their application nor do they get the money to upgrade it [**lierschDeveloperThreats**]. Developers make a living of their applications. When they do not make a profit, or even lose money with their servers, they can no longer continue. The result is a loss of creativity, ideas and skill for the ecosystem.

2.1.2 Dangers for Users

The loss of developers in the ecosystem is bad for users, but they can also be harmed by piracy. Users prefer pirated applications, they seem to be free of charge, but that's not necessarily true. The application might be altered in different ways, e.g. malware may be included. The user will not notice it right away, since these *features* often happen in the background without their knowledge. The application may for instance start using an expensive service, like premium SMS, or upload the contacts to the internet without the user recognizing. Even if there is no malicious content implemented, the application can suffer from bad stability due to manipulated code and missing updates when related from an unofficial source. In general, the risk is very high that pirated software offers a user experience worse than the original. [**bitdefenderPlagiarism**] [**lierschDeveloperThreats**]
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 [**piracyRate**]. Sources for pirated applications can be easily found on the internet. A simple search, containing *free apk* and the applications name, returns plenty of results on Google Search. The links direct to black market applications, like Blackmart [**blackmartStore**], and websites offering cracked Android Application Package (APK), such as crackApk [**crackApk**]. 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 [**apksfree**]. Offering these applications is only possible when the license mechanism is cheated. Software pirates practice professional theft and expose users to risks (see section **??**).
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. [**xdaPiracy**] [**developersPiracy**] 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 [**recodeMonument**].
Android applications are at an especially high risk for piracy because of their use of bytecode, which is an easy target to reverse engineering as shown in the further proceeding.

2.2 Android

Android is an open source OS launched in 2007 and today mainly maintained and developed by Google. It is based on the Linux kernel and mainly targets touch screen devices such as mobile devices or wearables. The system is designed to run efficiently on battery powered devices with limited hardware and computational capacity. Android's main hardware platform is the ARM architecture, known for their low power consumption, but also runs on MIPS and x86 processors. The following gives an overview over the architecture of Android and a deeper insight in the runtime system of Android. The architecture of the Android software stack can be seen in figure **??**.

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

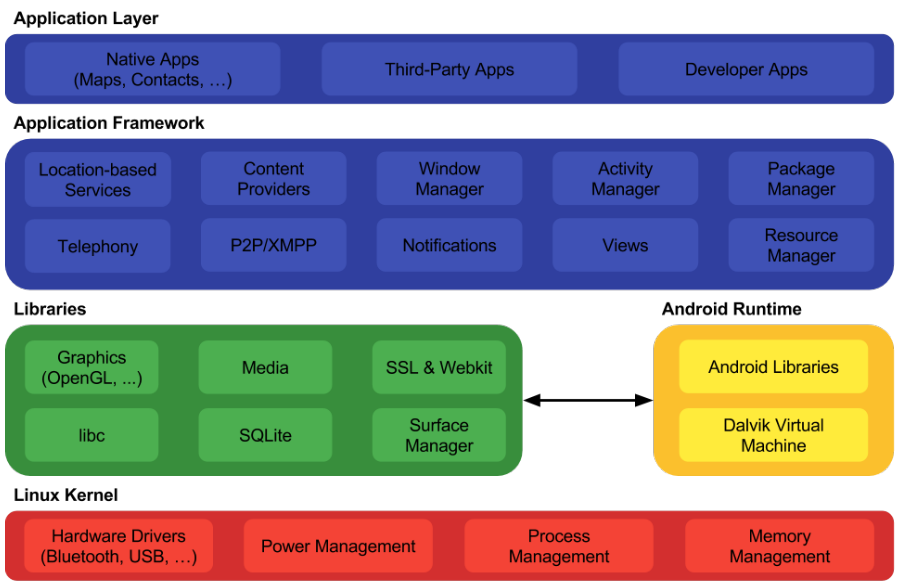


Figure 2.2: Android's architecture [**androidStack**]

management and controls the device drivers.

The layer above the kernel contains the Android RunTime (ART) as well as the native libraries of the system. ART and its predecessor, the Dalvik Virtual Machine (DVM), will be covered in section **??** and in section **??**.
On top of the libraries and the runtime lies the application framework. This layer provides generic functionality to applications over Android's Application Programming Interface (API), such as notification support.
The top layer is responsible for the installation and execution of applications.
Using these abstraction layers allows Android to run on a wide range of devices with different hardware and.

2.2.1 Android Application Package

Android applications are distributed and installed using the APK file format. They can either be obtained from an application store like Google Play, or downloaded and installed manually or by using Android Debug Bridge (ADB), from any other source. The APK format is based on the ZIP file archive format and contains the code and resources of the application.
The build process of APK contains several steps which are visualized in figure **??**.
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 **??**. 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.
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 and AndroidManifest.xml, containing compiled resources respectively essential information as required permissions

The *apkBuilder* combines these files into one archive file.
Before releasing the application, it has to be signed and zipaligned. The *jarsigner* is

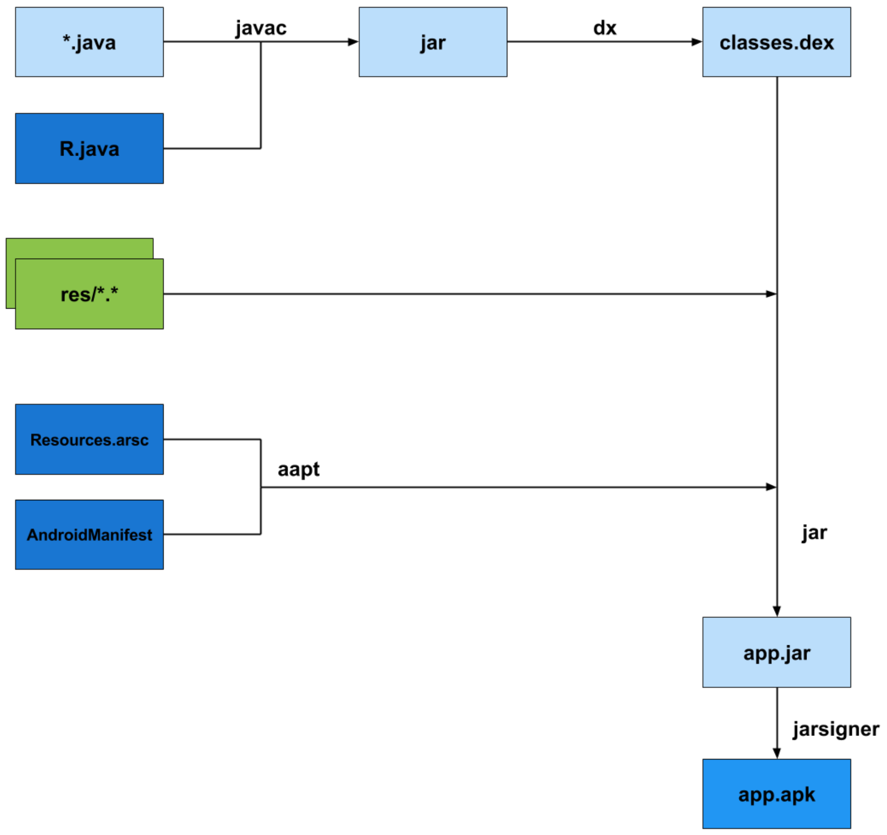
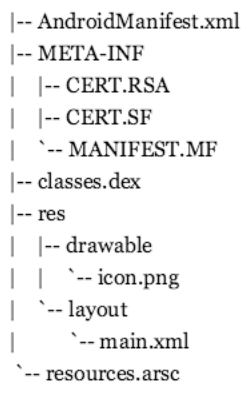


Figure 2.3: APK build process [**andevconDalvikART**]

used to sign the application.

It is done similar to the Java code signing [**codeSigning**] by adding a manifest, a signed manifest and the certificate. It is signed by the developer's private key and thus ensures the integrity and authenticity of the APK. Tampering can be detected and updates for applications with the same name and certificate can be installed. Afterwards *zipalign* is used to mark uncompressed data. [**androidPublishSign**] [**androidSigning**] [**andevconDalvikART**]

The structure of a final APK file has the following content as minimum seen in figure **??**. The *AndroidManfiest.xml* and the *classes.dex* have already been covered.
The *META-INF* folder is inherited from Java. It is used to store to store the signing information, e.g. the manifest and certificate [**codeSigning**] [**metaJava**].
While the static resources, like drawables and layouts, are in the res folder, the re-



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

Figure 2.4: APK folder structure

sources.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. [**kovachevaMaster**] [**ehringerDalvik**]

2.2.2 Dalvik Executionable File Format

As explained in subsection **??**, Android applications are distributed using APKs. They contain the application code as Dalvik bytecode that has been compiled from the Java source code.
Dalvik bytecode is suited to run on the ARM architecture. It supports direct mapping from dex registers to the 32 bit registers of the ARM processor. The opcodes are 16 bit, which makes the dex bytecode less compact than Java bytecode with its 8 bit instructions. The 16 bit opcodes result in 218 valid opcodes which have a source-dest ordering for its arguments. The complete instructions with arguments are 32 bit multiples. In case the instruction is larger than 32 bit instructions, adjacent registers are used to store it. [**androidDalvik**]
Similar to Java bytecode, values are not stored inside the method but as a reference pointing to the location. While in Java each class has its constants, like numbers, strings and identifier names, grouped together in heterogenous pool (see figure figure **??**, left side), Dalvik bytecode uses one pool for each type. When compiling Java bytecode to Dalvik bytecode, the heterogenous pools of each Java class are merged into one

global pool for each type (see figure figure **??**, right side). The merging of pool allows to remove duplicates and point their reference to only one value. This reduces the memory need for constant but increases the number of references. This approach is most effective for strings. A decrease of the memory footprint of up to 44% compared to the .jar is possible.
The compiled .dex file has the the structure seen in figure **??** on the left. Most important

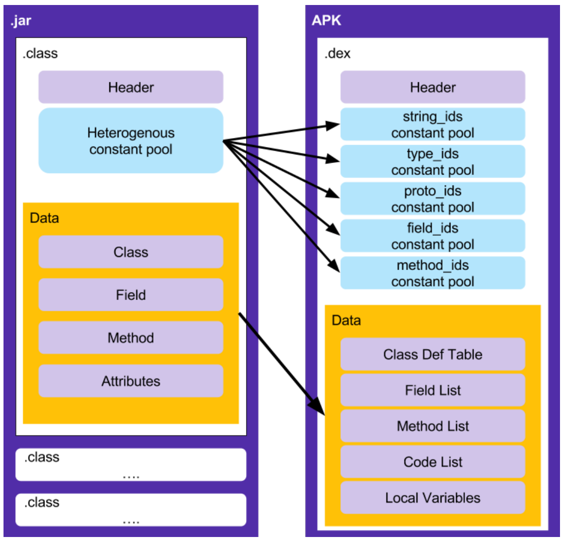
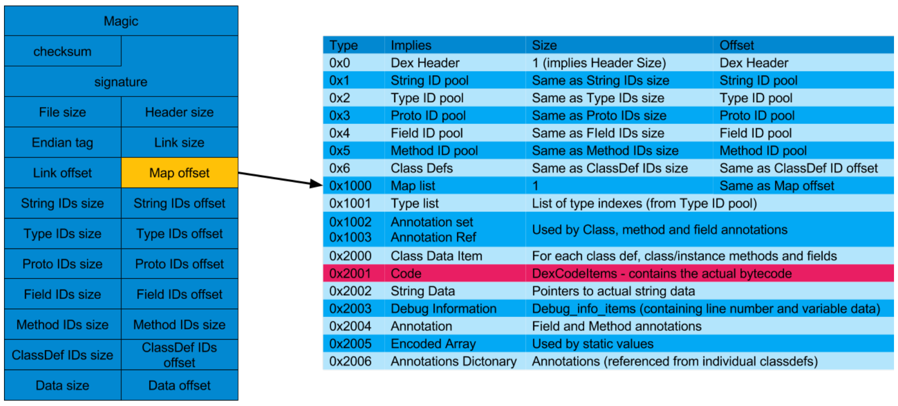


Figure 2.5: .jar to APK transformation [**googleDalvik**]

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 everything 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. It is based on every field except the magic field, checksum field and itself. The file can be uniquely identified by the signature. It is part of the file signing and stored in the manifest files to ensure integrity and authenticity. When the file is modified, both values have to be recalculated and updated. [**developersDalvik**] [**ehringerDalvik**]
Dex bytecode supports optimization. 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, .dex bytecode has a serious flaw. Since bytecode is pretty simple and contains a lot of meta information, decompilation can be done and the result is



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

Figure 2.6: .dex file format [**andevconDalvikART**]

easily understandable. At the same time, protection is rarely applied by the developers. This makes these applications an easy target for reverse engineering.

2.2.3 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 **??**). The differences will be explained in the following subsections.
Before initiating the installation, the APK is checked for its integrity and authenticity. The *META-INF* folder contains the necessary files for performing the signature verification process. The three files are the *MANIFEST.MF*, the *MANIFEST.SF* und the *CERT.RSA*. The manifest, *MANIFEST.MF*, contains the name and the digest of each file inside the APK, e.g. *classes.dex* and its signature. The the signature file, *MANIFEST.MF* which is used for the code signing. It is similar to the manifest and contains the SHA1 digest of the manifest and the digests of each entry inside the manifest. The *CERT.RSA* is the digital signature, called signature block, which is used to sign the files. While the manifest files are used to detect tampering while the digital signature is used to identify the code signer. In order to apply updates, the old and the new APK need to have the same package name and code signer. Since Android allows self-signed certificates, no connection of whether the APK was signed by the right person nor whether the application is allowed to be installed can be closed from the digital signature [**codeSigning**] [**androidSigning**]
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, the .odex is an optimization tailored to the specific architecture of the device for the best performance due to the the high diversity of Android hardware and their different processors. This is done once on installation. Future application starts will execute the .odex file instead of the the .dex file. This preprocessed version of the application has an improved startup time. [**kovachevaMaster**]
Currently, the Android runtime of choice is ART. For this runtime, the second step is more complex since the bytecode has to be compiled an additional time. This will be explained closer in section **??**.
After the bytecode is optimized respectively compiled, the application can be run.
When the application is run on the device, Android creates a sandboxed environment

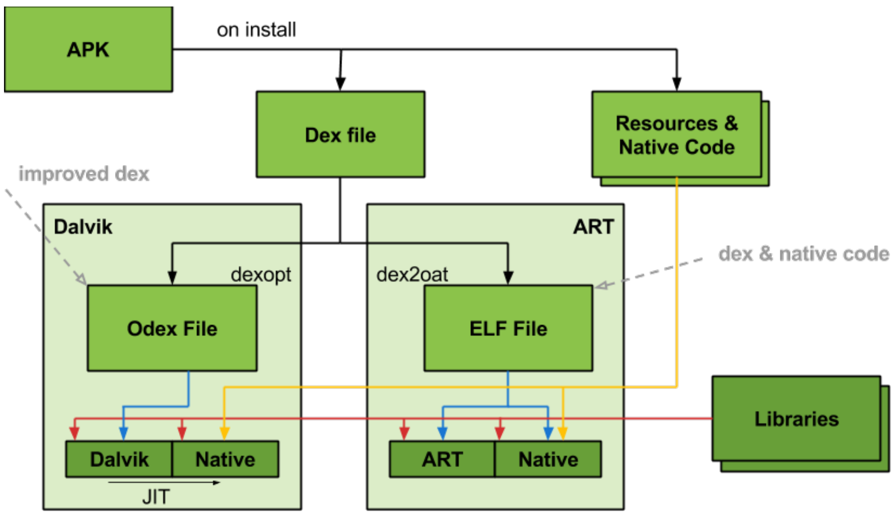


Figure 2.7: Installing an APK on a device [**googleIOArt**]

for application only. This means each process an own VM and separate user ID and cannot access any resources except their own. [**developerFundamentals**]

2.2.4 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 [**developersRelease**].
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, it has to be designed according to these requirements.
The DVM is a customized and optimized version of the Java Virtual Machine (JVM) and based on Apache Harmony. Even though it is based on Java, it is not fully J2SE or J2ME compatible since it uses 16 bit opcodes and register-based architecture in contrast to the stack-based standard JVM with 8 bit opcodes. The advantage of register-based architecture is that it 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 [**vecShare**] and uses zygotes. [**ehringerDalvik**] [**andevconDalvikART**]
The last big change made to the DVM was the introduction of Just-In-Time (JIT), which will be part of the discussion in subsection **??**, in Android version 2.2 "Froyo".

2.2.5 Android Runtime

In Android version 4.4 *Kitkat* Google introduced ART, designed to replace the DVM. It was optional first and only available as a preview through the developer options.
For backwards compatibility, ART still works with bytecode in the .dex files format [**androidArt**]. With the release of version 5.0 *Lollipop* ART it became the runtime of choice since DVM had some major flaws. Throughout the Android 6.0 *Marshmallow* previews it was constantly evolving and sometimes breaking compatibility with older versions. In addition, almost no documentation was available.
ART is designed to address the shortcomings of the DVM.

- expensive VM maintenance - wasteful JIT and too many CPU cycles by background threads
- frequent hangs and jitters caused by the Garbage Collection (GC)

This can be directly translated to increased battery usage and slower performance. The 32 bit support of the DVM look like a disadvantage because iOS already supports 64 bit, but it is not.
The improvements in ART make the maintenance less expensive, like replacing JIT with Ahead-Of-Time (AOT) and reducing overhead cycles. The GC is also non-blocking now and and can run parallel in fore- and background.
The main idea of ART and AOT is to compile the application to one of two types, either native code or Low Level Virtual Machine (LLVM) code. Each of the types has its

purpose and advantage. The native code runs directly on the hardware it was created for and offers improved execution performance while the LLVM code offers portability. In practice the preference is to compile to native since adding LLVM bitcode adds another layer of complexity to the architecture.
Different from DVM, ART uses not one but two file formats. Similar to the zygote of DVM, ART offers an image of pre-initialized classes and related objects at run time, the boot.art file. It is poorly documented and still changing a lot. The boot.art file is mapped in memory before the linked .oat file It is mapped to the memory upon zygote startup to provide improved application starting time . In addition to the boot.art file, there are two different .oat files. The boot.oat contains around fourteen of the most used Android framework .jars. The other .oat files are the former .odex files. They are still located in the Dalvik cache, but they are now Extensible Linking Format (ELF) files with the odex file embedded. Instead of *dexopt*, *dex2oat* is used to create these files. [**andevconDalvikART**] [**developersConfigureArt**] [**androidArt**] [**intelArt**]
In general, there is still room for improvement since not all code is guaranteed to be compiled. Since the base code is still dex and thus the VM is still 32 bit, ART is not fully 64 bit. The generated code is also not always as efficient as from a native compiler. The ART compilation process is likely to be improved as ART evolves.

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. Vulnerabilities which can be exploited to gain privileges are quite common. As seen in figure **??**, there have been over 30 vulnerabilities making it possible gain root rights since February 2015. Details and references of OS vulnerabilities can be accessed on pages like Common Vulnerabilities and Exposures or similar [**cveAndroidPriv**] [**cveDetails**]. 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 [**wugfresh**]. Rooting is usually bundled with installing a program called *su*. Unlike it in UNIX it does not mean *super user* but

switch 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 phone is then nonfunctional and no software cannot be executed anymore, even the OS itself. [**androidpoliceRoot**]
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 installation folder on the phone. 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.

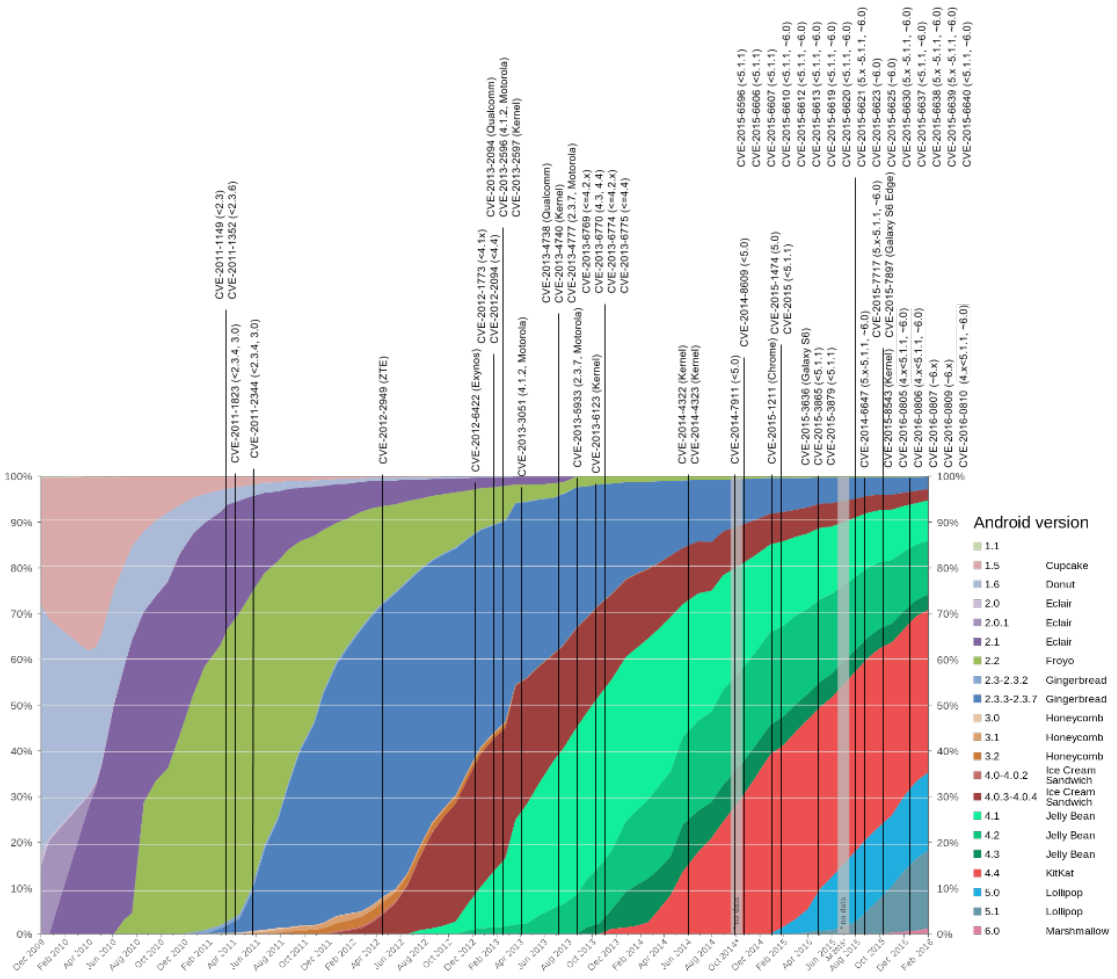


Figure 2.8: Timeline of market share of evolving Android versions and dates of root exploits recorded **[distributionRoot]** **[androidVulnerabilities]** **[cveAndroidPriv]** **[cveDetails]**

2.3 License Verification Libraries

Since the original copy protection of subsection **??** can be circumvented, a new and secure protection for Android applications is needed. This is not only relevant to Google, as the main of contributor to Android and provider of its biggest store, but also for other application store owner. Android philosophy is to allow the installation of apps from any source and not only the Google Play, other stores were created to get

a piece of Google's Android business. Some of the most widespread stores are from Amazon and Samsung.
Amazon does not only have the Amazon Store, but is also trying to create their own ecosystem by selling the *Fire tablets*. They use a flavor of Android tailored to fit Amazon's needs and come at a low price.
Samsung pursues a different approach. In addition to a store, they are also offering different services to bind to their ecosystem. There are different Chinese and niche stores as well.
The scope of this thesis are Google, Amazon and Samsung since they have a critical mass in users and a copy protection mechanism. Niche stores with less users and offerings are less subject to attack, while those without any copy protection do not even try to prevent piracy.
The stores in focus for this thesis have to fight the copy protection problem in order make their store attractive and attract developers by having low piracy rates.

2.3.1 Google's License Verification Library (LVL)

In order to tackle the copy protection problematic and to give the developer community a possibility to fight piracy, Google introduced the License Verification Library (LVL) on the 07/27/2010 [**developersLicensingBlog**]. It is easy to use and free of charge. The documentation can be found on the Android developers website [**developersLicensingOverview**].
Google's approach is based on a network service. It allows to query the trusted Google Play license server in order to determine whether the user has a valid license.
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 an unique implementation by changing its code, structure and control flow to a unique makes it less of a 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 app 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 app 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. [**developersLicensingSetup**]
After an App is registered with the Play Store and the set of keys has been received,

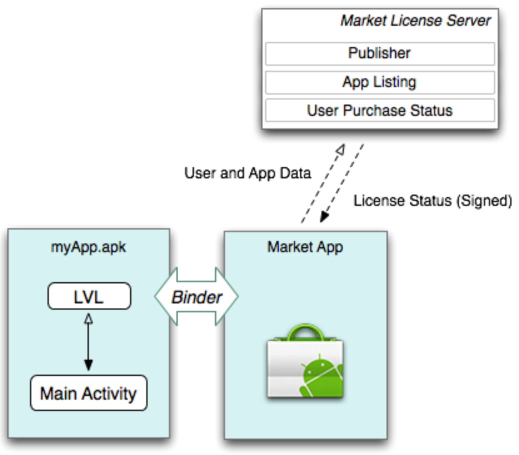


Figure 2.9: Google's implementation of license checking [**developersLicensingOverview**]

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. [**digipomLvl**] [**developersLicensingOverview**]
The first extension is the licensing permission in the *AndroidManifest.xml* (see code snippet **??**). 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. [**developersLicensingSetup**] [**developersLicensingAdding**]
The second extension is the implementation for the asynchronous handling of the

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

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

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 **??**. 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 re-

sult should be handled. **[developersLicensingOverview]** **[developersLicensingSetup]** **[developersLicensingAdding]** **[digipomLvl]**
The third extension is the license verification call which can be seen in code snippet **??**.

```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 license check callback

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. **[developersLicensingOverview]** **[developersLicensingSetup]** **[developersLicensingAdding]** **[digipomLvl]**
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

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

        mChecker.checkAccess(mLicenseCheckerCallback);
```

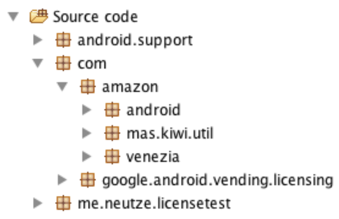
Code Snippet 2.3: Setting up the LVL license check call

username and other information and sends the license check request to the server. On the Google Play server, it is checked whether the user has purchased the application and a corresponding response is send back to the Google Play Service client. The response is encrypted to ensure integrity and detect tampering. The Google Play Service client passes it back to the LVL which decrypts the response, evaluates it and triggers callback accordingly. [**developersLicensingOverview**] [**developersLicensingSetup**] [**developersLicensingAdding**] [**digipomLvl**]
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.
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. [**developersLicensingOverview**] [**developersLicensingSetup**] [**developersLicensingAdding**] [**digipomLvl**]

2.3.2 Amazon DRM (Kiwi)

Amazon started its own application store in October 2010 [**amazonBeta**] as an alternative go Google Play. The Amazon appstore opened to the public on the 03/22/2011 [**amazonRelease**]. It can be used on Android and *Fire* tablets. The store comes with its own Digital Rights Management (DRM) since the Google LVL only works with the

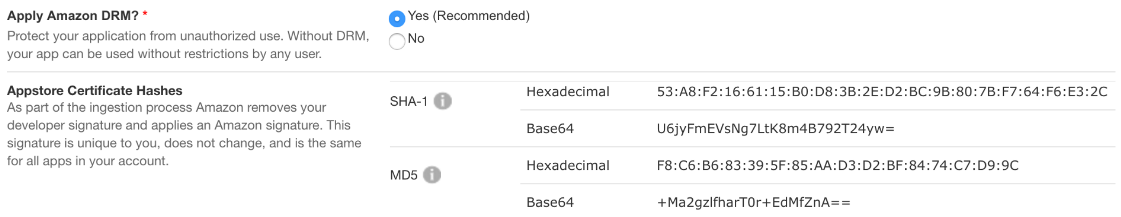
Google Play Store. The DRM is called *Kiwi* as seen in the reverse engineered code in figure **??**.
The prerequisites for using *Kiwi* are similar to the ones of the LVL, since the developer



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

Figure 2.10: Amazon library structure in decompiled application

requires a developer account on the Amazon Developer Service platform. According to the description, the library is to "Protect your application from unauthorized use. Without DRM, your app can be used without restrictions by any user." [**amazonDeveloper**]. 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 **??**). 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 **??**. Instead of using the developer's own key, Amazon uses a developer specific key. The information about the key can be retrieved from the developer platform as seen in figure **??**. [**amazonDeveloper**]
The implementation can be analysed with reverse engineering. The license verification



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 2.11: Developer preferences in the Amazon developer console [**amazonDeveloper**]

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 **??**. When the

application is launched, not only the application is initiated as before, but also the *Kiwi* DRM functionality is started by calling *Kiwi.onCreate((Activity) this, true)*.
The license verification works in combination with Amazon's Appstore application

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

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

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

2.3.3 Samsung DRM

Another major player in the smartphone business is Samsung [**comscoreMarket**]. 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* [**samsungZirconia**]. 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 [**samsungZirconia**]. 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. The developer is free where to implement the three code additions needed.
First of all, the required permissions have to be added to the *AndroidManifest.xml*. Zirconia needs access to the internet and to the phone state, containing the phone identifier IMEI and the phone number. The implementation can be seen in code snippet **??**.
The second addition is the implementation of the *LicenseCheckListener*. It contains the two results, either valid or invalid license verification result. While *licenseCheckedAsValid()* contains the code for success, *licenseCheckedAsInvalid()* is used when the license cannot be validated. . The third addition is initialization of the license check. Zirconia

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

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

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

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

Code Snippet 2.6: Zirconia license check callback

handles everything in its own. The developer just has to set the listener for the license verification callback and start the check by calling the *checkLicense()* method.
Zirconia always follows the same internal pattern when the license check is executed. First, it is queried for a stored license from previous verifications. If a stored license 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, acquired with the phone state permission, and the application. The server evaluates whether the user is authorized to use the application and replies accordingly. The response is unique for each device and application combination and thus cannot be used on another device. In case the access is granted, Zirconia stores the license on the device. The next time the license check is initiated, the same flow is done.

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

Code Snippet 2.7: Setting up the Zirconia license check call

2.3.4 Implementation Abstraction

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 **??**.

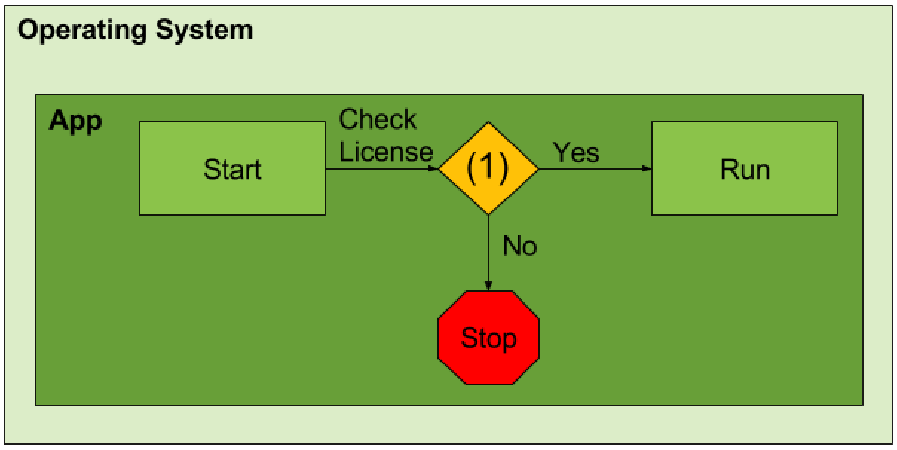


Figure 2.12: Abstraction of the current license verification mechanism. The library is represented by *(1)*

2.4 Code Analysis

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

2.4.1 APK Extraction

The analysis of the cracked code is performed using different tools on a desktop computer. The APK has to be extracted from the phone and transferred onto the computer, because the tools cannot analyse the application while it is still on the phone. This is done using the *adb shell* which is part of the Android SDK.
The *adb shell* can be used inside the computer's command line tool. The Android device must be connect to the computer via USB and USB debugging has to be activated in the device's developer settings. The *adb shell* is used to access the filesystem of the device 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 send to the computer.

Note: *Root* is neither a prerequisite to access the APKs nor to alter their *classes.dex* file.

2.4.2 Code Abstraction Levels

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

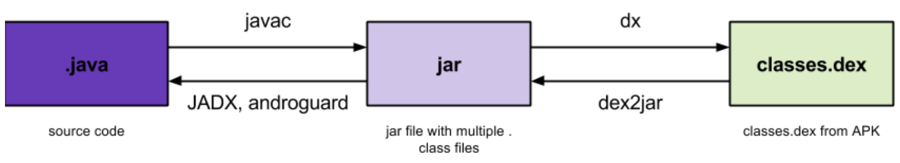


Figure 2.13: Java .class and .dex can be transformed bidirectional [**andevconDalvikART**]

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 in the bytecode that have been modified.
The extraction of the *classes.dex* is done using a simple script shown in code snippet **??**. The APK is an archive file and can be unpacked using *unzip*. The content is unpacked

```
#!/bin/bash
#hexdump dex
unzip baseapk -d /tmp/
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 **??** 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 [**opcodes**]. For example, the first 8 byte or 16 hex tuples, *64 65 78 0A 30 33 35 00*, are the .dex file magic, which identifies the file type. Translated to ASCII, the result is *dex.035.*.

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

Code Snippet 2.9: Hexadecimal view of classes.dex as classes.txt

Smali Analysis

The smali code is disassembled from dex bytecode using *baksmali* [**smali**]. 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[**smali**]. The syntax is loosely based on the Jasmin syntax [**smali**]. 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 **??**.
An example of smali code can be seen in code snippet **??**. It is easier to understand than

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

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

the dex presentation. The content of variables, as in line 3, can be identified without big effort. This enables the reader analyse the application's work flow similar to the source code.

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

Code Snippet 2.11: smali code example

Java Analysis

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 **??**, the .dex files are isomorphic to the corresponding Java .class files. This makes it possible to decompile the dex bytecode into Java code. The decompilation to the exact source code cannot be achieved since in the compilation process some information is lost. Another problem is the optimization for the DVM. The dex bytecode may contain Dalvik patterns that are unknown to the Java decompiler. The outcome is not always sufficient and for this reason two different decompilers, DAD and JADX, are used.
DAD, is a short name of "DAD is A Decompiler". It is part of Androguard [**androguard**], a reverse engineering tool for Android. It works with the dex bytecode and does not require third party tools like dex2jar [**dex2jar**]. Code snippet **??** shows how it is used in the command line interface to decompile the APK into Java code.
JADX [**jadx**] is the second decompiler. The decompilation is directly done from the

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

Code Snippet 2.12: Script to decompile to Java using androguard

APK's dex bytecode to Java code and can be performed in the command line interface as seen in Code snippet **??**.

```bash
#!/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 Code Comparison

The amount of code of the abstraction levels is huge. Most of the code base of an abstraction layer stays the same after the attack and the changes are only punctually. Diff a standard command line tool used to compare two files with little differences in an overall unchanged file.
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 **??**). Doing this automatically and using diff saves a lot of time.
The result does not only contain the change as original and new code, but also the

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

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

location where the change happened. The example diff of a dex file is presented is code snippet **??**.

```
@@ 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
```

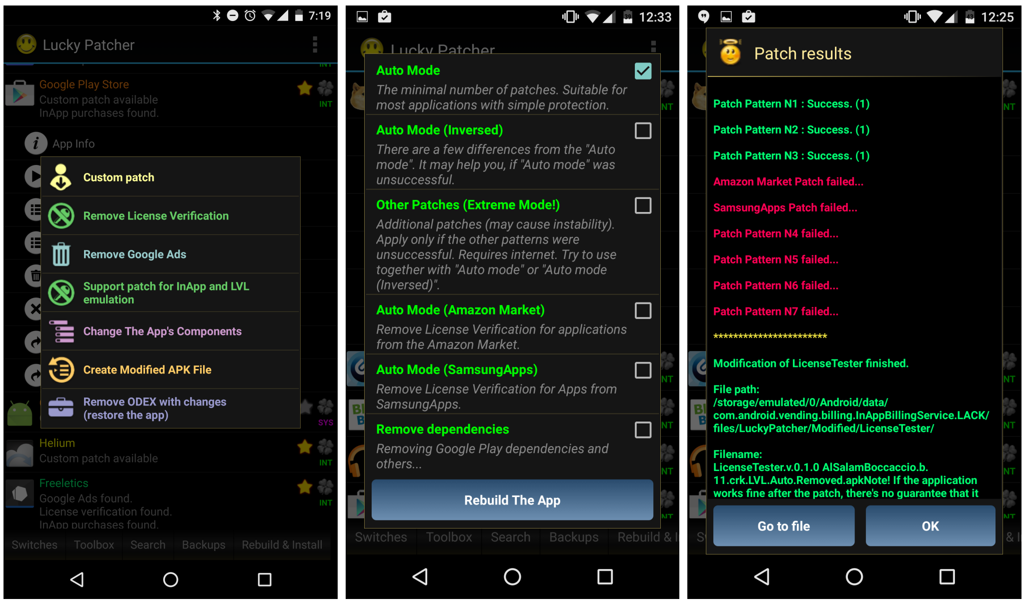
Code Snippet 2.15: Diff on Dex level for N1 pattern

3 Cracking Android Applications with Lucky Patcher

The cracking of applications on Android is a widespread phenomenon these days since the modification of the dex bytecode is possible. There are a number of tools voiding the license verification library of premium applications. Many developers discuss this piracy threat and the tools used on the internet .
This thesis will focus one of the most popular cracking application, 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" [**luckyPatcherOfficial**]. It is written by a developer calling himself ChelpuS and currently on version is 6.0.7 (03/03/2016).
Lucky Patcher offers different modifications which can be used to alter functions of an application. They can be applied to remove the license verification in premium apps or Google Ads and to change or restrict permissions and activities [**luckyPatcherOfficial**].
Lucky Patcher requires no technical knowledge and offers automatic cracking for non professionals. This combination makes it a popular and an effective tool with a high potential for creating damage. [**munteanLicense**]
The application can be downloaded as an APK from the official website [**luckyPatcherOfficial**] and installed on any device, *root* is not 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. In order to unlock all features of Lucky Patcher, the device has to be rooted.
When an application is selected, a submenu offers various actions, e.g. to get information about the app or run it. The patches menu opens the submenu of figure **??**, on the left. It shows the available patches for this application.
There are two types of patches. The first type is the *custom patch* and the second type are universal patches. Their goal is the disabling of the license verification libraries, removing Google Ads and rebuilding of the application to use the emulated LVL and



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...

```
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
```

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

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

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

corresponding store.
There are six automatic modes available as seen in figure **??** 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 **??** on the right. It is indicated that different patching patterns are used to void the license verification. The different patching patterns are analysed and explained in section **??**. /newline Lucky Patcher does neither guarantee that all license verification related restrictions are removed nor that the application works at all.

As described in section **??**, the license verification is implemented as client-server connection. The server communication is secure against man in the middle attacks or spoofing, as messages are encrypted [**munteanLicense**]. Lucky Patcher is taking a different path by modifying the application itself. A black box analysis is made to identify what the strategy targets and alters.

3.2 Blackbox Analysis

The blackbox approach looks at the parts of the code manipulated by Lucky Patcher. Lucky Patcher is used on different applications to get a variety of results and identify how the different implementations are attacked. This is done with applications implementing the three libraries in the scope of this thesis, LVL, *Zirconia* and *Kiwi*.
The applications are patched using the *Auto Modes*. The approach for creating a modified APK was chosen since .odex files are device specific and cannot be used for a general conclusion of an attack (see subsection **??**). The outcome is analysed using the methodologies described in section **??**. Since only the bytecode is modified, a static analysis is sufficient.
The goal of reverse engineering the code and comparing it to the original application is to pinpoint changes and analyse them on different abstraction levels. This includes the .dex level, on which Lucky Patcher works, the smali level, which makes the .dex code human readable and the Java level, as the highest level of abstraction and human readability. On each level, the modified and original code are compared. The utility diff is used to retrieve the changes in an easy way, ignoring the unchanged code.

Besides circumventing the Google LVL, Lucky Patcher supports the the voiding of Amazon's and Samsung's license verification library. A reference application with known source code and an implementation, according to the tutorial of LVL

[**developersLicensingAdding**], is created in order to have a reference application. This application is called *LicenseTest*. It gives full control and knowledge of the code and allows the analyse of Lucky Patcher attack on the most comprehensive version. The same application, including the Google LVL, is uploaded to the Play Store, Amazon and Samsung. Of course, for Amazon and Samsung, the LVL has to be disabled because it is not supported on those platforms. For Samsung, *Zirconia* is implemented according to subsection **??** while Amazon injects *Kiwi* as discussed in subsection **??**.
To see how Lucky Patcher handles different implementations, other applications than Lucky Patcher were analysed as well. The applications, are Runtastic Pro[**runtasticApp**], version 6.3, and Teamspeak 3[**teamspeakApp**], version 3.0.20.2, for the LVL and A Better Camera [**abettercamera**], version 3.35, for the Amazon DRM. These apps were chosen since they were already owned and approved to be included into the thesis by their developers. They include Google's LVL and Amazon DRM. The analysis for Samsung is done by using only one application. The .jar of *Zirconia* cannot be changed by the developer and thus has to be implemented the same way into all applications.
Run time tests were done on the author's device as well as several others in order to proof LP's ability to target different configurations, e.g. Android version or *root*. This test is necessary since the modified application can be installed on any device. It verifieds whether the crack works even though the corresponding store, root or internet connection are not available.
As described before, Lucky Patcher offers different modes to patch applications. Each mode offers a set of patterns to be used of which each change a piece of binary. These patterns are shown in figure **??** on the right. A pattern is identified by a name (N1, N2, ...) and consists of a set of predefined sequences of bytecode in which certain values are modified. For the black box test each mode is applied in order to discover patterns applied.
These are the different modes and what Lucky Patcher describes them as.

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

- Auto Mode (SamsungApps) - "Removes License Verification for Apps from SamsungApps" (Note: SamsungApps is called GalaxyApps, see subsection **??**)

3.3 Patch Patterns

In order to identify the changes done by each individual *Patch Pattern*, the code of the original and cracked APK are compared. The changes are inspected on dex, smali and Java level with the tools explained in section **??**.
The names Lucky Patcher assigns to the *Patch Patterns* and Patches are taken from the patching result output in figure **??** on the right. 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 modus. Table **??** gives an overview based on the black box test. It shows the patching modes and the *Patch Patterns* or *Patches* applied.
As shown in the following, the changes observed in the black box test are only

	Patch Patterns									
Modus	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 3.1: Overview of *Patch Patterns*/*Patches* applied by each mode

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 to be changed 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, 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* and appear in the context of the *Patch Pattern* they belong to. Each *Patch Pattern* can have multiple bytecode patterns, 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. 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 colored in green and red, fixpoints in blue and placeholders 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 as *0a* instead of hexadecimal values like *0x0a* is chosen for improved overview in the diff files. When converting .dex files to smali files, the arguments of the opcodes are transferred to variables, e.g. x in dex code is vx in smali.

When the dex code of an application is modified, the checksum has to be recalculated and the file has to be signed again. These changes can be seen in the diff of the dex files but not explicitly shown in this text. They are required by Android to make the .dex file valid but do not change the application logic.

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 [**developersLicensingReference**].
The *Patch Pattern* swaps *1a* and *0f* in their order, as seen in the dex code in code snippet **??**.
No matching bytecode pattern for *Patch Pattern N1* is identified in the source code analysis of Lucky Patcher, neither by name nor by verification.

```
@@ 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 variables can be identified as blocks of a switch statement. Due to the internal mapping by the language, variables have different names. The swap of switch cases *0x1* und *0x2* can be seen in the diff of code snippet **??**.

```
@@ 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 **??**, 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. Despite the input, it always handles it as if the user is verified.

Patch Pattern N2

Like *Patch Pattern N1*, *Patch Pattern N2* is applied in all patching modes, except the solo extreme mode. It is more aggressive, since it does not only attack the LVL library, but extends it attacks to other Google Mobile Service (*gms*) libraries, e.g. *com/-google/android/gms/ads/*, as well. The extended analysis of different applications shows custom libraries are attacked as well. An example is AnjLab's inapp billing library [**inappBilling**] of FKUpdater, located at *com/anjlab/android/iab/v3/Security*. The library contains code for the Google in app billing. Other locations are targeted in order to find a moved LVL. Similar to *Patch Pattern N1*, *Patch Pattern N2* targets the *LicenseValidator* class's *verify()* method.
The changes in the .dex file can be seen in code snippet **??**. 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*. Lucky Patcher uses the placeholder *S1* for the second bytecode. The target remains the same while the source is modified. The placeholder indicates the the source is set to *1*. One bytecode pattern, consisting of the search and replace pattern, is found in the code of Lucky Patcher.

```
@@ 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 **??** names the target opcode. Instead of moving the result of the preceding function to variable *v5*, it is always 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 **??**) 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 the result of the signature validation is ignored. *verify()* is continued as if the signature was valid.

Patch Pattern N3

Patch Pattern N3 is different than the other *Patch Patterns*, since it is applied with opposing goals. The first version, *Patch Pattern N3*, is used in auto mode while *Patch Pattern N3i* is used in the inversed auto mode. The name *Patch Pattern N3i* i is chosen since it is used in the *inversed* mode. LuckyPatcher groups both *Patch Pattern* as *Patch Pattern N3* since they both attack the same part of code inside the classes defining the policies. In case of the basic implementation of the LVL, the *APKExpansionPolicy* and *ServerManagedPolicy* in the *com/google/android/vending/licensing/* folder are attacked. The two classes are the policies examples provided by Google [**developersLicensingReference**]. The *Patch Pattern* targets the *allowAccess()* method inside these classes.
The *Patch Patterns* have an opposing result. While *Patch Pattern N3*is replacing the *01* with *11*, *Patch Pattern N3i* does the opposite by replacing *11* with *01* (see code snippet **??**).
The source code of Lucky Patcher contains three categories of patterns. The first category are the four bytecode patterns for *Patch Pattern N3*. It replaces the target instructions arguments with the the placeholder *S1*, i.e. the sources of the target instructions are set to *1*.
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.
The third category are bytecode patterns which cannot be assigned to one of these categories.
While the four patterns of *N3* and *N3i* have the same context but different variables, the two patterns of *N3x* are different in their structure but enforce the same logic as theother patterns. The first bytecode pattern of *N3x* targets implementations using *13*, a move for a wide constant. In this case the variable *W0* is set depending on *R0*. The second bytecode pattern of *N3x* is applied when the integer value is retrieved from an array (*44*). The instruction is replaced by *12 10* and a *00 00*.

```
@@ 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

@@ 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

@@ 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 3.8: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N3* and *N3i*

When looking at the smali diff in code snippet **??**, 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 3.9: Diff on Smali level for *Patch Pattern N3* and *N3i*

The resulting Java code is shown in code snippet **??**. The *Patch Pattern* enforces 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

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

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

Code Snippet 3.10: Diff on Java level for *Patch Pattern N3* and *N3i*

the verification result is according to the policy or not. Both variables, which can be returned as result, are initialized with the same value. This makes the result independent of the outcome of the verification. Two approaches with opposing result are available since the the accepted return value can be easily inverted by changing *true* to *false*.

Patch Pattern N4

Patch Pattern N4 was applied once in the test sample and does not target the basic implementation. It is part of the auto and auto inverse patching modes. The target of the *Patch Pattern* is the *LicenseChecker* class of the LVL. It is responsible for initializing the license check in its *checkAccess()* method [**developersLicensingReference**].
As seen in code snippet **??**, it replaces *38* with *33*. The one bytecode pattern found in Lucky Patcher's source code shows that the arguments of *33* are enforced to be *00* instead of the original byte, which are variable..

```
@@ 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 3.11: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N4*

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

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

Code Snippet 3.12: Diff on Smali level for *Patch Pattern N4*

The Java code in code snippet **??** 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()*. Since the result of the method is identical, if called twice, and thus 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 3.13: Diff on Java level for *Patch Pattern N4*

Patch Pattern N4 ensures that the result of textitcheckAccess(), whether the policy allows to continue, is never considered.

Patch Pattern N5

As part of the extreme mode, *Patch Pattern N5* targets the *LicenseValidator*'s *verify()* method. It cannot be applied on the standard implementation of the LVL.
The diff of the dex code in code snippet **??** shows the replacing of *0a* with 12. The location of the change is dependent the eight fixpoints of the search pattern mask. *Patch Pattern N5* makes use of three bytecode patterns with different context, found in the source code of Lucky Patcher.

```
@@ 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 3.14: Diff on dex level and search and replace bytecode pattern to apply for *Patch Pattern N5*

The bytecode modification by the *Patching Pattern* is identified change from moving a result (*move-result*) to moving a constant (*const/4*) to the variable.

```
@@ Patch Result N5 @@
```

```
- move-result v1
+ const/4 v1, 0x0
```

Code Snippet 3.15: Diff on Smali level for *Patch Pattern N5*

The result of patching can be seen in the Java diff in code snippet codeSnippet:n5DiffJava . The original code parses the response code from the response data. After applying the patch, the response data is still parsed but the result is ignored and the response code is set to *0*.

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

Code Snippet 3.16: Diff on Java level for *Patch Pattern N5*

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

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* changes three bytecode tuples of the .dex file which can be seen in code snippet **??**. The first changed value is *38* which is replaced by *12*. The second value is *06* which is replaced by *00*. The third change is the replacing of *4a* by *00*. All three changes are applied by a single bytecode pattern.

```
@@ 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 3.17: 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 **??**. The first results in the initialization of *p2* with *0*. The second change is required to get a valid syntax. The opcode *const/4* has two bytecode tuples while the original *if-eqz* has four tuples, 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 3.18: Diff on Smali level for *Patch Pattern N6*

The changes of the *Patch Pattern* are presented in code snippet **??**. 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 3.19: 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 is the implementation for the

callback for interprocess communication and receives the async response from the license server [**developersLicensingReference**]. In addition to the lvl classes, the *Patch Pattern* is applied to all classes residing in the `com/android/` package.
Patch Pattern N7 replaces *0a* with *12* by applying the seven bytecode patterns that are available in the source code of Lucky Patcher. Each does the same change but targets a different context.

```
@@ 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 3.20: Diff on dex level for N7 patch

smali blabla

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

Code Snippet 3.21: Diff on Smali level for N7 patch

The outcome might not be stable anymore and thus should only be applied when the other modes are not successfully .
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.

```
@@ Patch Result N7 @@
```

```
- this.verifyLicense(data.responseCode, data.signedData, data.signature);
+ data.responseCode;
+ this.verifyLicense(LICENSED, data.signedData, data.signature);
```

Code Snippet 3.22: Diff on Java level for N7 patch

The goal of this attack is to patch the *onTransact*, wherever it may be located. This methods is responsible for calling the *verifyLicense()* implementation of the *ILicenseResultListener*. This way, the server's response code is ignored and the code for *LICENSED* (*0*) is passed.

Amazon Market Patch

Lucky Patcher's attack on the library of Amazon does not have different *Patch Patterns*. Since the *Kiwi* library is injected by Amazon and cannot be customized by the developer, only one patch is necessary. The bytecode pattern is applied twice while 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* works similar to the LVL's *Patch Pattern N4* and replaces (*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 3.23: Diff on dex level and search and replace bytecode pattern to apply forthe *Amazon Market Patch*

The patch replaces *if-eqz* with *if-ne* as seen in code snippet **??**. 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 3.24: Diff on Smali level for the *Amazon Market Patch*

The Java code presentation helps to interpret the changes of the attack. Since the if statement is now always wrong the code block for response codes not *APPLICATION_LICENSE* is never called. The same canges are applied to the d.java class. Analysis of the dependencies shows the function checks whether the given string is not null and then returns *true*. After patching, the result of vo.equals is always true always *true*and the entire if-condition is always false.

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

Code Snippet 3.25: Diff on Java level for the *Amazon Market Patch*

The analysis of the *Amazon Market Patch* indicates that there are less patterns needed since code modifications have to be expected. Patches are applied to manipulate the checks in case the response code is different than *APPLICATION_LICENSE*. The result is forced to be always *true* and thus the license verification always passes.

SamsungApps Patch

Similar to the Amazon's *Kiwi* library, Samsung's *Zirconia* library cannot be modified. For this reason cracking the library requires only one patch. The *SamsungApps Patch* is applied on the *LicenseRetriever* and *Zirconia* class in the *com/samsung/zirconia* package. It uses two *Patching Patterns*, called *S1* and *S2* in order to distinguish between them. *Patch Pattern S1* is applied on both classes once and *Patch Pattern S2* is applied twice but only on the *Zirconia* class.
While *Patching Pattern S1* replaces *d6* with *00*, *Patching Pattern S2* uses *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

@@ 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 3.26: Diff on dex level and search and replace bytecode pattern to apply forthe *SamsungApps Patch*

The result of *Patching Pattern S1* is that the *if-eq* statement compares uses *v0* and *v0* instead of 2 different variables. The result of this comparison is always *true*.
Patching Pattern S2 has the effect that *v0* does not return the result of the preceding 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 3.27: Diff on Smali level for the *SamsungApps Patch*

The presentation in code snippet **??** contains the changes in Java code. Instead of checking the response code for validity, *LicenseRetriever*'s *receiveResponse()* method always skips the check, when *Patching Pattern S1* is applied, and executes as if it was valid. 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 @@
- return foo();
+ return true;
```

Code Snippet 3.28: Diff on Java level for the *SamsungApps Patch*

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

3.4 Conclusion and Learnings

The summary of the LVL patterns and their use in the patching modes can be seen in table **??**. Amazon and Samsung are always successful since not the developer implementation is attacked but the library itself, which is always the same.
When patching Amazon, Samsung and the LVL with the auto and inversed auto mode, the patches are applied determined to the important parts of the application. They are effective as long as the library is not modified by the developer. In contrast to the determined patching of the automatic modes, the extreme mode tries to apply patterns to files in other locations as well as to more complex methods. This might cause instability, as seen in pattern N6, since it alters the syntax of the dex file.

Table **??** gives an overview of patched applications. Lucky Patcher does not guarantee to be successful with one or more patching modes.

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

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

The first example is *LicenseTest*. While circumventing the license verification with the auto mode alone and combined with the extreme mode is possible, it does not work with the other modes. One reason is due to the inversed auto mode tailored to the opposed configuration of the *allowAccess()* method implemented. Another reason is that the extreme mode is an addition to counter modified license verification implementations.
The second example is *Runtastic Pro*. This implementation is altered in a bad way which makes it vulnerable to all patching modes. In the pirated and unmodified application, the user is told that the application is under maintenance. When the patches are applied, the user can login as if the application was bought in the store.
The third example is *Teamspeak 3*. Their interpretation for the license verification is able to withstand all attacks.

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

The changes are applied by extracting, modifying and repacking the *class.dex* similar to the build process of subsection **??**. The modification of the file entails that its checksum and signature have to be recalculated. Since Android can detect tampering of the APK, the digest of the file in the *META-INF* folder and its manifest files has to be adjusted as well [**androidSigning**]. Lucky Patcher does not have the developer's private key to sign the files accordingly. This causes problems when certificate is checked for its origin but

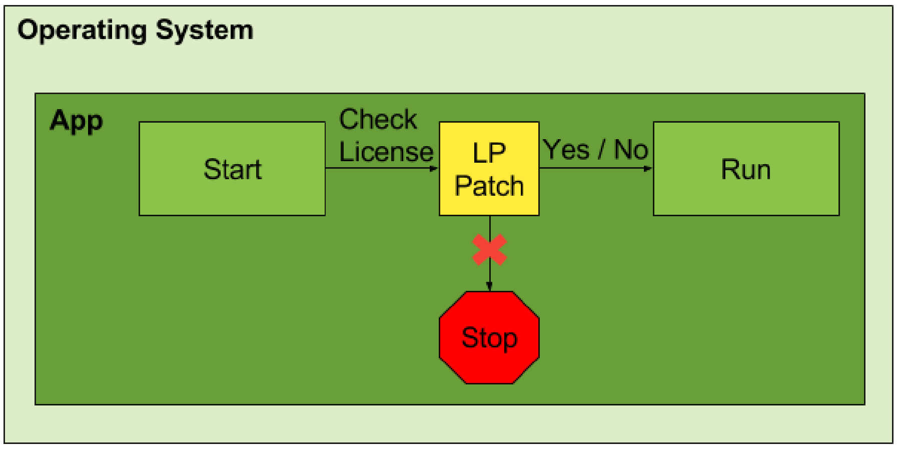


Figure 3.2: Abstraction of the current attack on the license verification mechanism

since Android allows self-signed certificates it does not matter [**codeSigning**]. Android just compares the certificates in case an application is updated with a APK containing the same package name or when trust relationships between applications are about to be established. [**androidSigning**] The only limitation is the installation of the modified APK is only possible when the original application is removed and future updates are not possible as long as the new APK contains the same certificate.
The creation of the modified APK does not require *root* and the cracked APK can be distributed without limitations. This makes piracy easy for users and is a huge threat to developers.

4 Countermeasures for Developers

Now that that the methodology of Lucky Patcher's attacks is analyzed, suggestions to prevent it are proposed.
The first chapter will cover methods to improve the current implementation of the LVL, additional integrity tests to detect tampering and suggestions for a implementation using native code..
The second chapter proposes methods no longer dependent on *yes/no* decisions, but the availability of content. /newline The third chapter takes an outlook on ART to tackle the shortcomings of dex bytecode regarding the license verification.a

4.1 License Verification Library Extension

The first suggestion is to fortify the spots attacked by Lucky Patcher and identified in section **??**. The goal is to prevent the application of the bytecode pattern and thus the success of automatic patching. This is done by changing the bytecode of the application, i.e. by modifying the LVL implementation. Additional checks are introduced and stop execution in case tampering was detected. Since it is possible to alter code manually after analysing the it, obfuscation is introduced as a tool to make reverse engineering more time consuming.

4.1.1 Modifications on the Google LVL

Attacking with Lucky Patcher is often successful because many developers do not customize the LVL at all. One reason for not customizing the LVL is that protection against casual piracy is sufficient for the developer. Casual piracy is the copying of APKs from one device to another by uninformed users. /newline Another reason is that they do not know where exactly to fortify the library and thus they do not want to spend additional effort [**developersSecuring**].

This thesis presents two approaches to fortify an application against attacks by Lucky Patcher.
The first approach is to actively go against Lucky Patcher's patterns by modifying the identified parts of code.

The second approach is to implement the LVL with native code which is not targeted by Lucky Patcher's auto patching modes.
While the idea of the approaches is valid for all license verification libraries, the 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.

Unique Implementation

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 of the license verification mechanism. 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.[**developersSecuring**]
There are three areas the developer should focus on when modifying the LVL [**developersSecuring**].

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 **??**, these two classes are the primary target of Lucky Patcher and thus should be altered as hard as possible while retaining the original function. Isomorphic code changes can include:

- replacing the switch statement with an if statement and add additional code between the if statements (see pattern N1)
- deriving new values for constants and check for these values in the further proceeding (see pattern N3)
- removing unused code for interfaces, e.g. implementing the policy verification inline in the *LicenseValidator* instead over an interface (see patterns N2, N4, N5 and N6)
- move the LVL package into the application (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

- implement 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. [**developersSecuring**]
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. [**developersSecuring**]
Attackers do not only target the LVL, but the handling of the result in the application. This can be prevented by handling the mechanism in a separate activity. This prevents from scenarios where the attacker voids methods which would prevent further proceeding, e.g. the In addition, the license verification can be postponed to a later point in time since attackers are expecting it on the the applications launch. [**developersSecuring**]
These modifications are easy to apply and make the implementation unique. There are unlimited ways to implement them. It needs to be stated, that they are as 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 gained knowledge 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. [**developersSecuring**]

Native Implementation

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

- The developers implements an its own native version of the LVL

- Google provides a native implementation of the LVL

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

- heavy obfuscation and encryption must be applied
- dynamic code generation and automatic customization every time it is loaded - having only one version is an easy target

In addition to making the license check native, parts of the application should be moved into the native code. This protects against attacks where the call of the license verification library is skipped.

In general, the proposal is simple, but the implementation is much harder. Until now, no one has come up with such universal approach for the broad masses. This indicates that still a lot of research and work has to be done to implement this solution. As long as the the license verification library is implemented in native code, the automatic patching modes of Lucky Patcher do not work since they only target dex bytecode. Attackers have to crack the native library and offer it as a custom patch.

4.1.2 Tampering Protection

The modification of the application is only guaranteed as long as the integrity of the application and environment is ensured.
When circumventing the LVL the code has to be modified. There are different indicators for an attack on the application. Breaches of integrity are forced debuggability, *root*, an installed cracking applications or installation from a source different than the store.
Each potential breach of integrity can be tested and the program interrupted when in doubt. This should be done without any hint on the reason for the killing, as any reason can be used for reengineering purposes. Of course, this may result in a bad user experience. These tests are implemented as seen in figure **??**. Since all

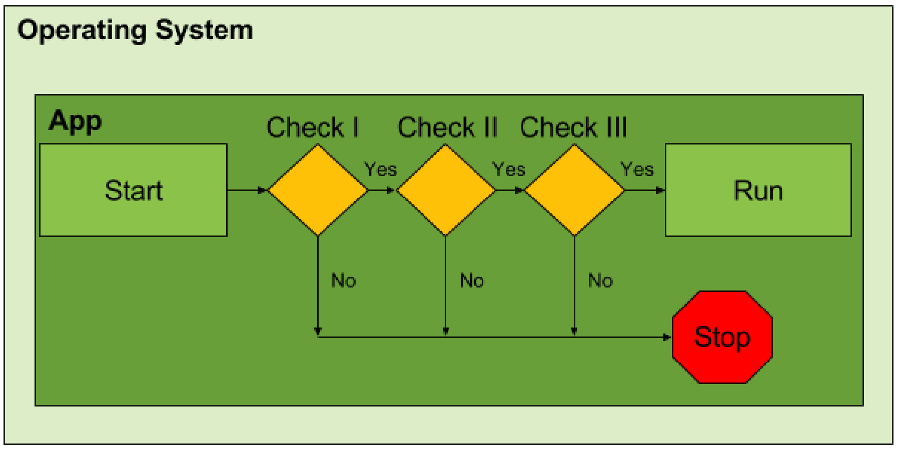


Figure 4.1: Introduction of additional tests to check environment and integrity of the application

tampering countermeasures have the *yes/no* schema, they can be circumvented easily. The goal of these checks is to increase the workload of an attacker. The code has to be analysed in order to find, understand and patch them. The checks can be spread inside the application to unexpectedly crash the application. The attacker not only has to invest time to figure out why the application crashes randomly, but also to find these checks. Obfuscated and implementing it randomly increases the effort as well. Even the possibility of implementing them in native code can be considered.
This does not prevent Lucky Patcher itself from working, but it offers an additional layer of security which has to be voided and seemingly random crashes with no explanation will be detrimental to the user experience.

Debuggability

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

```
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

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.
Google has introduced a similar APKt, called SafetyNet [**safetynetGoogle**]. It is used to check the “health and safety of an Android"[**safetynetDev**]. It is said to be used in security critical applications like Android Pay and the reason for excluding rooted devices from the service [**safetynetGoogle**] [**safetynetPay**] [**safetynetPayx**].
Since *root* is achieved by the *su* file in the filesystem, the application can search for its existance in the common locations. In case the search is successful, the execution of the application can be terminated.

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

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

        return result;
    }
```

Code Snippet 4.2: Example code for checking for root

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 [**androidCrackingTools**]. The check is more specific as the *root* check as it only excludes people who have a piracy tool.
As shown in code snippet **??**, the check tries to acquire 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 app will not run, if Lucky Patcher is present.

Sideload

As modified APKs can be created and installed on the 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

```
    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

Android versions already include a library called *AntiPiracySupport* [**antipiracy**]. It has a similar goal and blacklists and disables pirated applications.
The code snippet **??** shows the implementation for the stores in scope for the thesis. Additional stores can and should be added in case the developer decides to offer the application in another store. In case he does not, the application will not work when retrieved from a non listed store.
This feature should be implemented with caution, since Google notes that this method relies on the *getInstallerPackageName* which is neither documented nor supported and only working by accident [**developersSecuring**].

Signature

Application code is signed to authenticate an application developers and enable them to provide updates for the application [**androidSigning**]. Since the signature has to be

```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

rewritten when cracking the application, it is an indicator for attacks [**tamperSignature**]. 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 [**maps**].
The code for the signature check can be seen in code snippet **??**. In order to check the application's signature, the original signature has to be provided (see line 53). The application's signature fetched from the package information (line 57ff). In case the signature cannot be retrieved or the signature do not match, the check terminates the application.

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

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

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

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

        return result;
    }
```

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

4.1.3 Obfuscation

The first steps to fortify the LVL have been made. The library is modified and the environment's integrity is checked. This helps against Lucky Patcher's automatic patching modes, but is still vulnerable to manual attacks and Lucky Patcher developing and mutating. Android applications are at high risk of being reverse engineered, as its bytecode is easily decompile. In order to make reverse engineering and prevent the development of custom patches, obfuscation is introduced.
An obfuscator is an easy to apply protection and should be used in every application. The obfuscator can either be applied to the source code or bytecode. There are open-source and commercial Java obfuscators available that are also working on Android, e.g. *ProGuard* [**proguard**]. Some dex obfuscators exist as well, like *DexProtector* [**dexProtector**]. Three obfuscators are explained in the following below.
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. [**developersSecuring**]
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. [**developersSecuring**]

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.

Dexguard

DexGuard is a commercial Android obfuscator for Java code. It is considered the offspring of *ProGuard* and specialized on Android. The provided methods are a super set of *ProGuard*'s ones. [**strazzareLevel0**]
In addition to the obfuscation, *DexGuard* offers application and platform integrity, resource protection, communication hardening and code protection [**dexguard**]. This way, the application is not only fortified against code analysis but also reverse engineering. The side effect of using *DexGuard* is that the application becomes smaller and the performance is increased by the smaller memory usage.

Allatori

Allatori is a commercial Android obfuscator from Smardec. Similar to *DexGuard* it provides a superset of *ProGuard*'s methods. [**allatori**]
The features of *Allatori* include name obfuscation, string encryption and debug information obfuscation. Even flattening of the structure and obfuscation of the control flow are possible. Another feature is the addition of complexity to algorithms, e.g. loops are modified in a way that reverse engineering tool do not recognize them as such. [**strazzareLevel0**] [**allatoriDoc**]
Overall, the resulting application has a decreased .dex file and memory footprint while having increased speed.

4.2 Content Driven Application

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

4.2.1 Content Server

This approach is to fight the shortcomings of the license verification libraries by moving them to a server. Users have to login on a server in order to verify their license. Instead of returning just the result of the verification, the server delivers part or all of the content of the application. Since license verification no longer is dependent on an unary result, Lucky Patcher fails. The content provided by the server cannot be second guessed by Lucky Patcher.
The implementation can be described with the application Spotify [**spotify**] as an

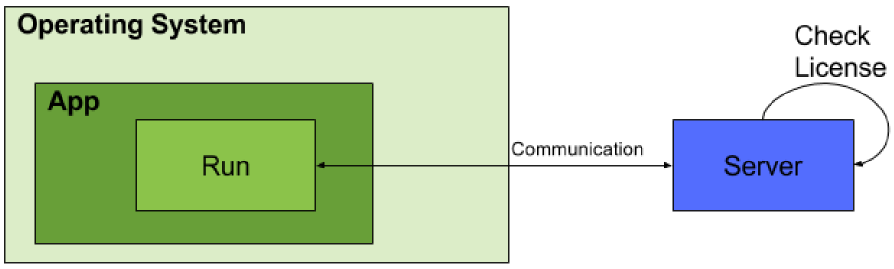


Figure 4.2: Abstraction of an application and a content server

example. Instead of verifying the license locally on the device, the user has to enter his credentials and send them to the server. In case the credentials are valid, the user is logged into the application. The content, the music in this case, is no longer on the phone itself, but streamed from the server. The attacker still can circumvent the login process inside the application by manipulating the code. Since the content is on the server and the user has to be authorized on it, no content is available inside the application. Thus attacks on the applications itself do not work anymore.

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. This means that it must be possible to extract parts of the application's logic and implement them on a server. This is not possible for all applications.

The second problem are the additional resources needed. When outsourcing parts of the application on a server, not only money is needed for the server, but an additional application for the server has to be created. Not every developer can handle this additional workload.
The third problem is the always online necessity. It limits the freedom of users and creates additional internet traffic. This is not accepted by all users.

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

4.2.2 Encryption

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

Encryption

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

Encryption - Resources

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

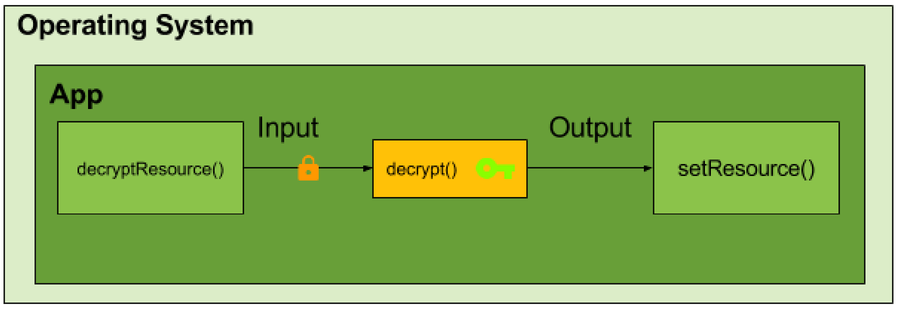


Figure 4.3: Encrypted resources have to be decrypted before they are used or displayed

Encryption - Action Obfuscator

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

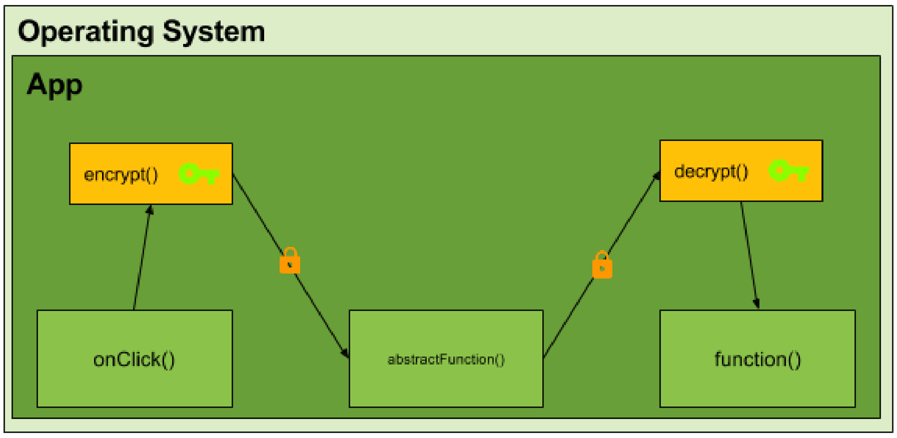


Figure 4.4: Encrypted actions to obfuscate dependencies

Encryption - Communication

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

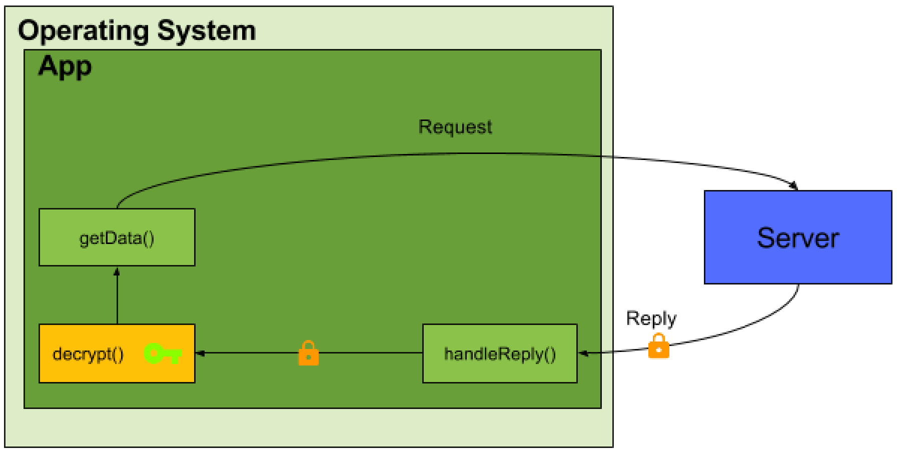


Figure 4.5: Encrypted communication with a server

Key Storage

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

Key Storage - Online and Caching

The first approach to handle the encryption key is to store it on a server and provide it to the application. This works similar to the license verification.
On decrypt method call, the application tries to retrieve a cached cryptographic key. In case a cached key is available, the application starts to decrypt the content. Otherwise the key is requested from the server. The server does a verification of the user similar

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

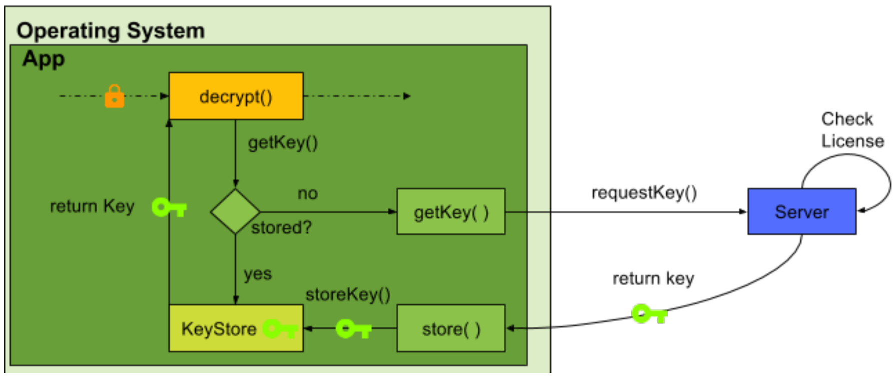


Figure 4.6: Retrieving the key after successful identification from the server and store it local on device

Key Storage - Secure Element

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

advantage of an SE is that its functionality is outside of the Android application and thus cannot be manipulated by Lucky Patcher.
An abstract presentation of the use of a SE can be seen in figure **??**.
At the moment, the integration of a SE comes with some problems.

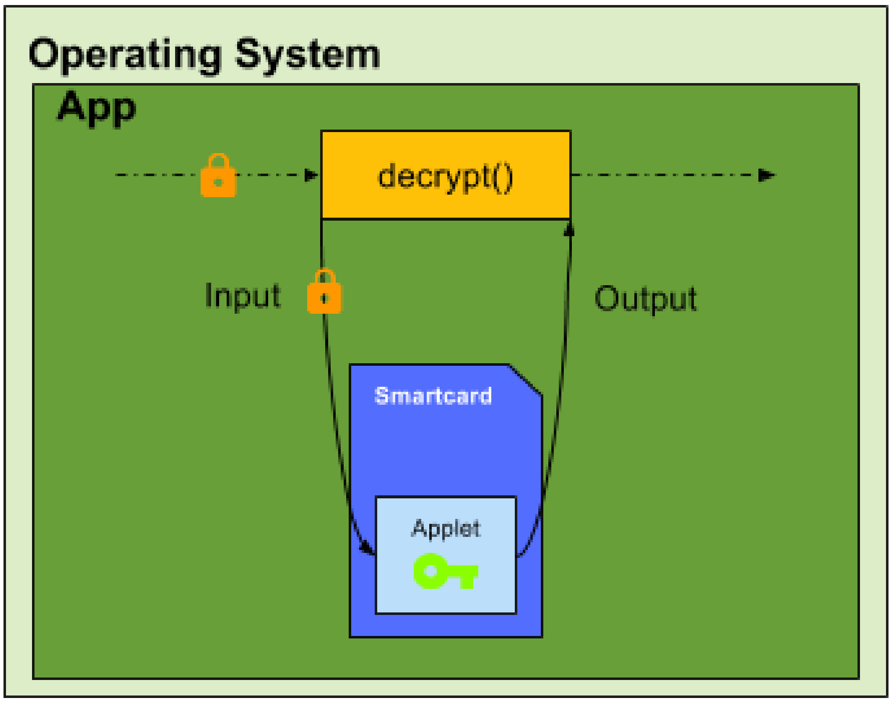


Figure 4.7: Decryption by using a smartcard

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

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

se signiert mit key+android_id welche unique ist

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

4.2.3 Key Management

4.3 Android Runtime

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

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

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

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

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

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

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

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

5 Conclusion

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

5.1 Summary

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

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

5.2 Discussion

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

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

shortly after start insufficient reilience against reverse engineering, not explusivly to lvl thus shift from lvl protection to general protection against reverse engineering, decompilation and patching

eternal arms race no winning solution against all cases, jsut small pieces quantitative improvement no qualitatively improve resilience limited to quantitative resilience, matter of time until small steps generate more work for reengineering, ggf lower motivation for cracker only matter of time until patching tools catch up, completely new protection schemes need to be devised to counter those [**munteanLicense**] research and also a valuable market for companies
Because source code can be easier recovered from an application in comparison to x86, there is a strong need for code protection and adoption of existing reverse engineering methods. Main parts of Android application functionalities are realized in Dalvik bytecode. So Dalvik bytecode is of main interest for this topic [**schulzLabCourse**] not a question of if but of when bytecode tool to generate the licens elibrary on the fly, using random permutations and injecting it everywhere into the bytecode with an open platform we have to accept a crack will happen [**digipomLvl**]

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

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

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

close down free installations

Es muss generell immer abgewogen werden zwischen Reichweite und Sicherheit. Von Output den Lucky Patcher gibt, sind die auto patching modes für Google, Amazon und Samsung, die großen Player. Ein Developer muss seine App dort anbieten um

Aufmerksamkeit zu bekommen. Deswegen sind diese Stores auch so gut "maintained" von Lucky Patcher. Im Falle, dass ein Developer "Sicherheit" vorzieht und seine App in einem alternativen Store anbietet, gibt es zwei Scenarien. Entweder entwickelt jemand einen Custom Patch (dex oder native Angriff) wenn ein "allgemeines Interesse" besteht oder die App ist uninteressant und erhält keine Aufmerksamkeit, weder von LP noch Kunden. Nur weil ein kopeirschutz nicht gekackt ist heißt es noch ncith dass er nciht knackbar ist, sobald genügend interesse besteht wird es jemand versuchen

5.3 Future Work

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

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

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

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

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

google hat schon sowas wie google vault

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

List of Figures

List of Tables

List of Code Snippets