INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation zum Erreichen des akademischen Grades eines
Dr. rer. nat. (Doktor der Naturwissenschaften)

Improving Copy Protection for Mobile Apps

Nils Timotheus Kannengießer

INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Improving Copy Protection for Mobile Apps

Nils Timotheus Kannengießer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende/r:	Univ. Prof. Dr. Claudia Eckert
Prüfer/in der Dissertation:	1. Univ. Prof. Dr. Uwe Baumgarten
	2. Prof. Sejun Song, Ph.D.

Die Dissertation wurde am 10.08.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.11.2016 angenommen.

“Copy protection is never perfect” [1]

Thomas Aura, Dieter Gollmann

1 Acknowledgements

First of all, I would like to thank my main supervisor Prof. Dr. Uwe Baumgarten, who was always available to discuss any open questions. He also provided many helpful hints during the writing of this dissertation. Ultimately, he gave me the required workspace and utilities as part of my job at TUM, which surely helped me in finishing this document within the recent years. During this time, I highly enjoyed my work as a Teaching/Research Associate at TUM in introducing students to Android, and working closely together on interesting projects with major industry partners from both Germany and the US.

Moreover, I would like to thank Prof. Sejun Song, PhD for the feedback he provided as well as his engagement in related research papers throughout these years. He also provided students and me helpful hints in paper writing. In addition, I'd like to thank him for the support during my international research visit in September 2015.

In addition, I would like to thank our industry partner, Giesecke & Devrient and in particular Mr. Rizvanovic and Dr. Sterzinger for their support and for providing me their MSC product. I would also like to thank all industry partners that are not mentioned or their provided and helpful inputs for this work.

Furthermore, I appreciate the helpful replies by Mrs. Dr. Weinl from the TUM library on my questions regarding the best quotation techniques.

Moreover, I want to thank all my students for helping me on various aspects of this work by investigating subtopics, implementing my proposed ideas or just for evaluating the solution ideas, while also providing helpful information used in this research work. A big thanks to all of them and in particular the following: Sebastian Schleemilch, Yixiang Chen, Magnus Jahnen, Marius Muntean, Michael Bichlmeier, Patrick Bernhard, Norbert Schmidbartl, Janosch Maier, Philipp Schreitmueller, Ioana Negoita, Ozan Pekmezci, Johannes Neutze, Lorenz Stadler, Hans Kirchner, Lucas Jaros, Nam Bui, Jochen Hartl, Arves Baus, Thomas Petting, Tomas Ladek, Florian Gareis, Aser Abdelrahmen, Mohamad Ayad, Shiffudin Al Masud, Gabriel Michels, Jonas Raedle, Vadym Strelchenko, Tuba Topaloglu, Nikolaos Tsiamitros, Felix Weissl and Konrad Weiss.

A special thanks is also extended to Kordian Bruck and Philipp Fent, who did not contribute to this research work directly, but indirectly by developing and supporting one of our most ambitious apps, the TUM Campus App that was used for the performed surveys of this dissertation in recent months.

Moreover, I would like to thank my family, including my brother Simon, as well as my parents, Fritz and Irmgard Kannengiesser, for their general support on my studies throughout all these years.

Last but not least, I would like to thank Nancy Lorenz for the feedback she provided regarding the improvement of the English writing styles in this dissertation.

Thank you everyone.

2 Abstract

English:

This dissertation identifies existing issues with major copyright protection mechanisms used on the Android operating system by Google for mobile devices like smartphones and tablets. First, the general problem of weak copyright protections used on major app stores is introduced, and the fundamentals on Android itself are presented to make the reader familiar with the operating system and reengineering of the apps themselves. Furthermore, related research topics are reviewed and discussed. A security analysis of possible protection methods highlights the current situation of existing solutions used to protect Android software from piracy these days, while possible solutions to improve copyright protection on Android using e.g., secure elements or native code, are analyzed as well. In addition, other practical and conceptual ideas related to e.g., secure elements and trusted execution environments, are introduced that have responsibilities to stakeholders like Google and hardware manufacturers and need to be honored. Moreover, improved solutions using native code are shown. Based on the presented ideas, several sample implementations have been developed and evaluated, and show a significant improvement to the existing solutions provided by Google and Amazon already. An outlook on further research possibilities is given as well.

German:

Diese Dissertation behandelt die Thematik von Kopierschutzmaßnahmen für mobile Apps mit dem Schwerpunkt des Betriebssystems Android für Smartphones und Tablets. Hierbei werden zunächst das Problem eines schwachen Kopierschutzes bei Apps in den großen App-Stores aufgezeigt, sowie ein Überblick über Android und das einfache Reengineering von Android Anwendungen selbst gegeben. Ebenfalls werden vorhandene Lösungen und Forschungen diskutiert. In einer Sicherheitsanalyse zu möglichen, aktuellen Kopierschutzverfahren zur Vermeidung von Softwarepiraterie unter Android werden vorhandene Risiken und Probleme genannt, wobei auch bereits Lösungsvorschläge unter Verwendung von, z.B. Secure Elements oder nativen Code, in die Analyse einbezogen werden. Zusätzlich werden praktische und konzeptionelle Lösungsideen mit Bezug zu Secure Elements oder Trusted Execution Environments vorgestellt, deren tatsächliche Realisierung in Abhängigkeit zu weiteren Stakeholdern (Google, Hardwareherstellern) steht. Ebenso werden Möglichkeiten zur Verbesserung mit nativen Code aufgezeigt. Im Zuge praktischer Evaluierungen wurden ausgewählte Methoden exemplarisch untersucht, deren Ergebnisse signifikante Verbesserungen beim Kopierschutz im Vergleich zu bestehenden Lösungen - von beispielsweise Google oder Amazon - erkennen lassen. Darüber hinaus werden weitere Ideen und Möglichkeiten für künftige Forschungsarbeiten aufgezeigt.

3 Assumptions

Target versions and available hardware

This dissertation aims to provide information for the recent Android versions using the ART VM (6.x). Therefore, methods that apply to older versions of Android are not presented in detail. For evaluations, only official Google-branded devices like the Nexus series were used and available. Smartphones by other vendors (e.g., Samsung S5, S7, etc.) were not available and were only examined in a theoretical way.

Research Group / Students' theses

The theses of students involved in the research of the author are not listed under related work, but are referred to as work done in the research group "we", while quoting them as usual. The topics, as well as initial ideas, were usually defined by the author of this dissertation and guided in the required direction, while requesting certain implementations based on the author's ideas like, e.g., the nLVL, or the analysis of Lucky Patcher.

Reader requirements

Even the fundamental section tries to cover many topics; this dissertation requires general knowledge on all computer science topics, particular IT-security in general and a basic Android developer's knowledge. A master's degree in either computer science or a related field is highly recommended.

Additional guidance by hardware designer suggested (NDA requirement)

Furthermore, the presented ideas using secure elements (SE) try to show general methods that may be used with hardware from several manufacturers. The specialties of the used security equipment (like the MSC by Giesecke & Devrient) are not reviewed, however, the hardware is assumed to be safe, and evaluations (e.g., side-channel attacks, etc.) are out of scope and not performed in this dissertation. Based on the used product, further support by the hardware manufacture is recommended and was not available upon creation of this dissertation regarding used hardware and software (cf. G&D's MSC). All information that is assumed to be protected by our NDA is either blackened or omitted.

References

In general, this work was created by taking the suggestions of TUM's Quotation Guide[1] in mind.

In addition, open questions were discussed with my first advisor (Prof. Dr. Baumgarten) and library employees (Mrs. Dr. Weinl) and led to the following, additional guidelines:

- Videos: Any statements made here are quoted as direct quotations with their sources closely attached, while mentioning "transcript by author" in a footnote.
- Sources mentioned at the beginning or end of a section represent reported speech for that whole section.

[1] https://mediatum.ub.tum.de/doc/1231945/1231945.pdf

- Single or short phrases (e.g. Secure Element) are not quoted. Instead, their sources may be found at the end of the particular phrase or closely attached to that section. Exception: Special terms defined by the author only.
- In cases of reported speech of reported speech, attempts were made to discover and mention the original source or to highlight the used source with its used sources at least, (e.g., [Sch] (based on [13])).
- Since it is less common to mention sources for abbreviations, only general sources are mentioned and most keywords are referenced in the work or commonly known anyway.

Moreover, the following methods were applied:

- Source codes are used without quotations for easier reading, and the used sources are mentioned below each source code table (e.g., based on [Sch]).
- The latin abbreviation “cf.” is used in a similar sense to “see”. It may also represent the source for a section, while providing further details on the discussed topic as well.

4 Publications

4.1 Recently published / topic-related

15th International Symposium on Ambient Intelligence and Embedded Systems, Heraklion, Greece, 2016
Talk/Paper, Authors: Nils Kannengiesser, Johannes Neutze, Uwe Baumgarten, Sejun Song
Title: "An Insight to Cracking Solutions and Circumvention of Major Protection Methods for Android"

15th International Symposium on Ambient Intelligence and Embedded Systems, Heraklion, Greece, 2016
Talk/Paper, Authors: Nils Kannengiesser, Yixiang Chen, Uwe Baumgarten, Sejun Song
Title: "Securing License Verification by using Native Code, Fusing Options and Indirect Method Triggering on Android"

Android Security Symposium
Vienna, Austria, 2015
Talk, Author: Nils Kannengiesser
Title: "Secure Copy Protection for Mobile Apps"
Content: Introduction and latest ideas on the topic and research results

13th International Symposium on Ambient Intelligence and Embedded Systems
Aveiro, Portugal, 2014
Talk/Paper, Authors: Nils Kannengiesser, Uwe Baumgarten, Sejun Song
"Secure Copy Protection for Mobile Apps"
Content: Further ideas on the topic and early research results

12th International Symposium on Ambient Intelligence and Embedded Systems
Berlin, Germany, 2013
Talk/Paper, Authors: Nils Kannengiesser, Uwe Baumgarten, Sejun Song
"Secure Copy Protection for Mobile Apps"
Content: Introduction of the idea and initial approaches

4.2 Former publications / in general

9th Intl. Conference and Workshop on Ambient Intelligence and Embedded Systems Geel, Belgium, 2010
Talk, Nils Kannengiesser, Sejun Song, Helmut Dispert
“Development of an Android smartphone application for surveillance systems employing Cisco video cameras”

8th Intl. Conference and Workshop on Ambient Intelligence and Embedded Systems, Madeira, Portugal, 2009
Paper, Authors: Nils Kannengiesser, Thomas Ladehoff, Thorsten Knutz, Helmut Dispert
"Implementation of a platform independent client software for the GO Bluebox System”

7th Intl. Conference and Workshop on Ambient Intelligence and Embedded Systems, Kiel, Germany, 2008
Paper / Talk, Authors: Nils Kannengiesser, Helmut Dispert
“Implementation of a Security System based on RFID and WSN technology"

5 Content

6 Introduction

This section should give the reader an introduction to the topic by providing a quick and summarized overview on fundamental knowledge for understanding the issues discussed in this dissertation. Furthermore, the actual motivation and problem statement is explained, and the different sections of the work are introduced.

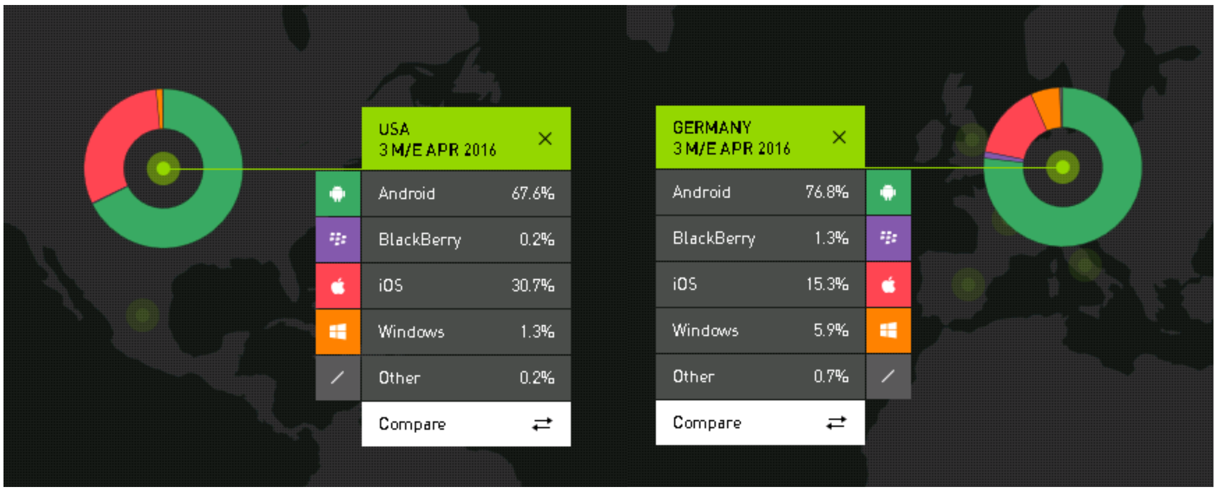
6.1 Topic introduction

6.1.1 History and market share of mobile devices

In recent years, smartphones have become essential tools for our daily lives. When Apple revealed the first iPhone to the market in 2007, it brought many advantages of modern desktop computers to a single mobile device [2].

Nevertheless, Apple Inc. was not the only company working on these modern phones, and as early as in 2003, Andy Rubin ran a company called Android with its focus on building software for phones and cameras. Google purchased Rubin's company in 2005, and started the actual development of today's most successful mobile operating system – Android [3].

The latest figures (see Figure 1) confirm the current trend that Android leads the market with differences based on the country, e.g., having a market share of 76.8% in Germany vs. 67.6% in the US (April 2016). In addition, one can see that Android is more frequently used in Germany than in the US, while Apple's iOS is more dominant on the US market [4]. Due to the high market share of Android, it is important to provide secure solutions in terms of copy protection to developers.



USA 3 M/E APR 2016	
Android	67.6%
BlackBerry	0.2%
iOS	30.7%
Windows	1.3%
Other	0.2%
Compare	

GERMANY 3 M/E APR 2016	
Android	76.8%
BlackBerry	1.3%
iOS	15.3%
Windows	5.9%
Other	0.7%
Compare	

Figure 1 - Comparison of market shares (US/Germany – April 2016) (based on screenshots from [4])

6.1.2 Typical app development and platform comparison of Android and iOS

While applications[2] are written for Apple devices using swift[3] [5], Android applications may be developed using Java or even C/C++, but Java is the preferred language to use [6].

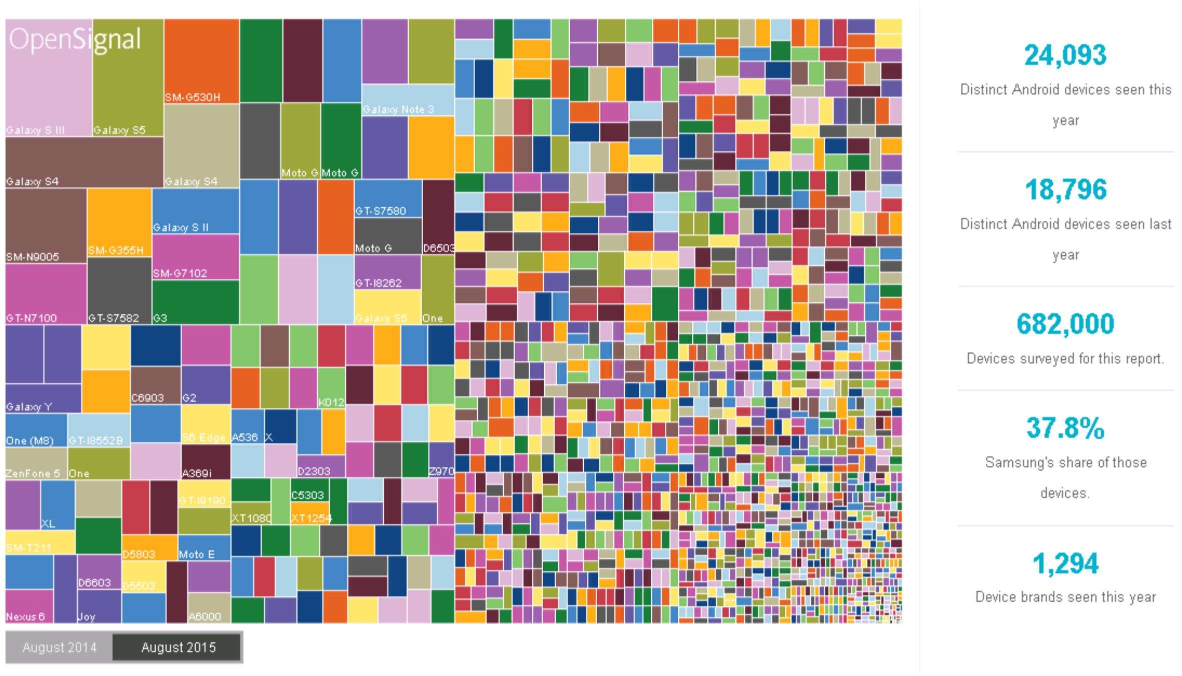
In general, app types can be categorized in native apps and non-native apps. Native apps are running on the devices themselves, while, e.g., WebApps can run in the context of a browser engine, or may be implemented as a browser frame for a mobile website only, and therefore, representing a very lightweight app to be started on Android. In addition, Android differentiates between native apps (developed in Java) and those that use native code (Java and C/C++), too [7] [6]. Furthermore, Android offers another specialty by executing apps on top of a virtual machine (see 7.3.2ff) [8]. Possible reasons for this approach are the platform independence of Java as well as security (cf. 7.3.2 / predefined permissions) and stability reasons in comparison to native apps with direct hardware access. For instance, the controlled access to resources also has proven to be a good approach in the past, and Microsoft integrated this approach in its operating systems for more stability (cf. “Built on top of the proven Windows NT Workstation 4.0 code base […] [it] adds major improvements in reliability” [9]). Nevertheless, virtualization does not always prevent malicious attacks (cf. exploits) as explained later [10]. Moreover, the usage of Java is beneficial to developers due to lots of existing frameworks that could be ported to Android quite easily. Ultimately, Java language is much easier to use and it provides simple to use interfaces to the hardware without complicated code requirements such as in C. In addition, Java avoids difficult debugging and lots of other issues (cf. pointers/segmentation faults, etc.). Moreover, Java is usually known by every computer science student (cf. TUM students) allowing Google and app developers to find developers and employees much easier.

Instead, Apple did not only require developers to learn C, but a completely new C-dialect called ‘objective-C’, or ‘swift’ nowadays [5], which is certainly a reason for several engineers to avoid that system. In the author’s opinion, it could even be another reason for Android’s success in recent years. Nevertheless, one of the fundamental difference comes with the selection of that language and apps for iOS are native apps and therefore better protected, resulting in far less piracy (e.g., 95% Android vs. 60% iOS for Monument Valley [11]), while so-called jailbreaking (rooting an iOS device) is not supported by Apple officially and more complicated, too. Apple draws a reasonable relation between app piracy and jailbreaking, since by default it is not possible to install apps from elsewhere, but the App Store itself, and Apple puts a high effort on the protection of copyright holders [12]. Instead, Android’s philosophy is different and the whole system is more opened, which results in disadvantages for the protection of intellectual property as outlined later in more detail. Another core difference between iOS and Android apps is that apps for Android may be developed and published within a day, and without the severe security checks like Apple does for their apps. Instead, Android customers are in charge of trusting an app, which sometimes leads to severe infections. In contrast, iOS developers need to wait several weeks for the successful completion of the audit of their applications [13]. Other, recent sources estimate that it takes 7 to 11 days for an app review by Apple [14] (based on crowdsourced data). The advantage of these severe checks is that Apple customer benefit from a more trustworthy App Store and ultimately, more credible apps; disadvantages may be that developers face longer waiting times and Apple may reject their app if it does not satisfy their

[2] also called apps on mobile platforms
[3] based on Objective-C

requirements, which can certainly be a burden. These requirements include insufficient information, apps with bugs, insufficient designed user-interfaces, misleading icons, or personal information requirements [15]. Instead, Android Developers have the advantage to design their apps in any way they like, and publish it almost instantly on Google Play. Nevertheless, over the recent years Google improved the requirements that developers need to add for a publication like privacy statements or mandatory screenshots. In addition, a design guide was released [16], but it is still up to the developers to make the decisions. For basic customer protection within Google's Play Store they use a service called "Bouncer" [17] that scans the market for malicious apps. Unfortunately, security researchers found loopholes quite early and as of today Google needs to update its service frequently to face new threats [18] (based on statements by Miller and Oberheide). In summary, one may say that Google's Android offers more freedom with certain risks, especially for new users, while Apple's iOS takes away the freedom in customizing the phone in every possible way, while providing their customers (including app developers) more security, which may be more suitable for beginners [19]. Nevertheless, in the author's opinion, Google tries to improve security more and more by locking the devices down (the operating system) and adding further restrictions with every release. This even had a negative impact on our solution (see 11.4.7); copy protection sometimes uses similar techniques as malware (see 9.1.1) and has similar goals of hiding its mechanisms.

6.1.3 Available devices on the markets



24,093
Distinct Android devices seen this year

18,796
Distinct Android devices seen last year

682,000
Devices surveyed for this report.

37.8%
Samsung's share of those devices.

1,294
Device brands seen this year

Figure 2 - Device Fragmentation by August 2015 (based on screenshots from [20])

Furthermore, Android is available to several devices with numerous different properties and is open-source[4] [21], while iOS is commonly known as being closed-source and customers are fixated in their choices of only a few available devices.

Google's own branded devices are the most well-known – the Nexus series (e.g., Nexus 5), is intended for developers and for keeping the balance between these expensive high-end devices

[4] exceptions apply, e.g., radio firmware

offered by Samsung or perhaps LG, while providing all new hardware and functions for testing purposes by developers. For that reason, all these devices can be rooted by default. Google also sells Android-based wearables and tablets [22]. In fact, its selling is not limited to developers, which is important in terms of the dissertation topic and rooted devices (as outlined later) are a threat to any protection at the moment.

In addition, many other vendors market their own smartphones these days, e.g., Samsung, LG, and HTC as one can find these products in every smartphone store. They integrate all kinds of additional features to fit the consumers' wishes in addition to their own apps and solutions on various topics. For instance, Samsung announced the "Trustonic for KNOX" [23] solution in cooperation with the Trustonic Company in order to enrich their devices with "enterprise and professional mobile security" [23], as Rick Segal[5] said. "Trustonic for KNOX combines the advanced and robust integrated security features of Samsung KNOX with Trustonic TEEs' hardware-based security to provide a trusted platform for service providers" [23].

In fact, Trusted Execution Environments (TEEs) may be the key to many issues and are also discussed in this dissertation, too (see 10.4ff). Nevertheless, TEEs are not available on all devices and require additional special hardware or cooperation. In general, one can identify [20] that we have several different devices with different sensors, screen sizes, and different Android versions. Figure 2 shows the device fragmentation based on data provided by users and collected by the OpenSignal apps for more than half a million devices [20].

6.2 Motivation (and security concerns)

"Android is insecure"

Providing secure solutions to most of these devices requires different solutions. Better security may affect numerous different areas from data protection to software protection. Right now, most applications face the issue of having no real private space. Many devices are often shipped in a locked state[6]; it is possible and even permitted by many vendors to unlock (and so-called root[7]) the devices as explained next (see 6.3.1), which is a real threat to data privacy and protection under certain circumstances. Besides the official permitted ways to root devices, it is possible to root phones by exploits. This method is also often used by malware.

The Security Bulletins by Google [24] as well as the CVE website filtered for Android related privilege-escalations [25] allowed us to gain insight on this severe threat of exploits for years. From the experiences in the past, it has to be assumed that there is a privilege-escalation exploit for any Android version soon after its release. Especially in 2015, Android faced extremely severe exploits that affected billions of devices and several versions of the "Stagefright" exploit [26], followed by further root exploits for millions of devices using Qualcomm chipsets as released by researchers recently [27]. Furthermore, as result of such a root exploit, an attacker can access any privately stored files by an app now, while it is possible to intercept and even manipulate the communication between apps and servers. This issue is covered in 10.1.5 for manipulating LVL communication, while fundamental knowledge about folder security is

[5] Vice President of Enterprise Business Team, IT & M.C. division at Samsung Electronics [23]
[6] limited access rights to folders and hardware (also commonly known as "not rooted")
[7] cf. relation to Linux's user with all system rights: root

introduced in 7.3.2. The application package file (APK file) can be received even without root rights for decompilation purposes (see 8.2ff).

“Insecurity supports software privacy”

A huge issue that derives from these security flaws is the effect on app- or service-sales as well as company secrets (e.g., hidden APIs). Software piracy is a huge problem in our modern world in general and not surprisingly so Android is affected here, too. According to a report by [28], developers try to adapt to the issue by offering free versions with buyable add-ons, while Google refuses to comment on that issue. This confirms a recent experience with Google by the author himself that Google showed little interest in supporting native copyright protection solutions for unknown reasons (see 11.4.8). Furthermore, in comparison to iOS, which is using native code apps, reengineering an Android app is fairly easy in most cases and requires skilled developers to avoid only basic issues already here (cf. 8.2 for more details on reengineering). Smaller companies especially might be affected quite hardly by software piracy due to their limited assets. The figures are sometimes dramatically, e.g., ustwogames released information that “Only 5% of Monument Valley installs on Android” [11] were legally bought, while the majority uses the game illegally [11]. Also, the developer of “Today Calendar” mentioned that about 85% of their users use a pirated version [29]. In the past other vendors (e.g., Epic Games) even decided not to release their games due to severe issues with software protection on Android as reported by Giga [30], while digital content companies like Netflix avoided Android at the beginning due to “the lack of a generic and complete platform security and content protection mechanism available for Android” [31]. In a recent move Microsoft announced (as reported by BR[8] [32]) the end for project Astoria that had the goal to port Android apps to Windows smartphones. According to that article, Microsoft made the decision due to IP[9] concerns by developers. While Google might have changed the situation for DRM protected content like movie-streaming by acquiring 3rd party technology [33], we still face the issue of an inefficient copyright protection mechanism on Android on the major app stores by Google or Amazon and their offered solutions for developers (see 10.1ff).

“Issues known and fixed in the desktop world”

One of the general issues derives from the usage of Java technology and its included references for cross-platform compatibility. Many of the current issues on Android are not new and solutions have already existed in the desktop world for decades (e.g., DashO Java Obfuscator [34]), but have to be adapted to the mobile world now. It appears that (especially at the beginning) Google did not focus on security that much, and this author recognized that integrated obfuscation solutions were not activated by default in recent years. In fact, it seems that many developers were and are not aware of the issues. Moreover, around 2013, we discovered some apps by major companies that were not protected at all, and this resulted in viewable hidden APIs or access codes. Examples of this might be the apps by BMW or games like “Worms” by Team17[10]. Only a slight margin of companies uses advanced protection tools like DexGuard [35].

[8] German TV broadcaster
[9] Intellectual Property
[10] Both companies were notified by the author and BMW even got in contact with the author to meet for detailed discussions and solution approaches immediately

“Copy Protections have a long history in the desktop world”

License Verification and Copyright Protection are well-known topics in the computer industry, and extend from the era of simple registration codes to specially prepared floppy disks, compact discs, and even later, downloadable software with online activation. In comparison, there are only a few of the technologies that can be seen on Android so far. For instance, in former days most copy protection techniques relied on simple activation, registration codes or type of a riddle that were shipped with a product (e.g., the PC game Monkey Island 2). Of course, this method was not that effective, and with the availability of inexpensive devices and mediums, companies were forced to look for more advanced approaches to protect their software from piracy. An example might be the usage of artificial sector errors on floppy disks [36] or later CDs. These artificial errors were difficult to copy for regular customers and required special manufacturing techniques or software at least. Famous protection techniques that should be mentioned here are SecuROM, StarForce and SafeDisc [37] [38] [39]. Over the years these mechanisms were improved by adding encryption, obfuscation, or other special attributes to make copying of protected software as difficult as possible [40]. Besides these software-related solutions, more expensive software products were often protected by so called (hardware-) dongles that provided a special reply to the software on request and are even used today, e.g., “USB-eLicenser” [41]. In the end, most of these techniques were cracked, disappeared from the market, or were improved.

Nowadays, most companies of the desktop world rely on encryption, online activation, are still using dongles [41], or simply force their customers to pay a monthly fee to be allowed to use the mandatory[11] server infrastructure in order to receive the actual game data, use the multiplayer option of a game, or communicate with friends (in a game). “World of Warcraft” [42] by Blizzard Entertainment is an example for such a game. Other vendors (e.g., Valve’s Steam [43]) provide their very own community and sales service platforms that offer lots of advantages against pirated software (e.g., ranking, gaming with friends, automatic updates, etc.[12]). Right now, this is one of the best approaches to get customers to buy a product, since it is simply not possible to use the complete product without a valid account (= paid product). Measurements that build on top of this requirement are cheating prevention systems like Valve’s Anti Cheat (VAC) [44]. It can be assumed that even however, there seems to be no public figures, this results in careful players trying to keep their accounts alive, while they benefit from its advantages (e.g., quick availability) as a former software pirate stated in a report by [45]. This report is also, where he acknowledged that he stopped software-piracy with the introduction of steam.

“Copy Protection on mobiles is in development”

Thinking about the mobile world and mobile devices (in terms of Android) it is still very different. We observed that the used techniques are often a few years behind the desktop-computer-era. This applies to the actual user interfaces of games or apps as well as to the used digital rights management (DRM) techniques (cf. security analysis in 10.1ff).

For now, smartphone apps or games are often - not always – played/user by a single user and most app content is shipped with the initial application, while the protection mechanism are based on rather simple protections like the License Verification Library (LVL) by Google or

[11] playing e.g. these games without its servers is not an option
[12] author’s experience

Amazon’s DRM (see 10.1ff for details) [46] [47]. However, it ultimately relies on the developers and their skills. For instance, some apps revealed the default implementation of Google’s LVL like the gaming app “Worms” by Team17, while other vendors have already thought about ways to improve it by renaming some packages or variables. It is also recommended by Google to add modifications [47]. Nevertheless, it has to be assumed that Google leaves copyright security once more to the third parties. As compared to Valve’s steam platform, it would be more reasonable that Google as well as the hardware manufacturer, take care of it and provide customers, as well as developers, with a better solution. As outlined previously and experienced by the author himself, Google shows little interest in this area so far.

Summary

In general, we can sum this up by stating that consumers must be careful in choosing apps for installations (cf. danger of malware), while developers have to be cautious to implement necessary security techniques. Otherwise, they will face the general issues with software piracy such as mainly lost revenue or maybe even worse, the sale of cracked, “’piggybacked’ apps” [13] [48] by criminals.

While attackers might be average users that use tools like “Lucky Patcher” [49] (see 10.1.3 for details) to crack certain apps for leisure purposes, we also face the increasing issue of organized crime, who repackage and redistribute apps with malware, and thereby gain money for exchanged commercials or for illegal sale [50]. More information on their reasons for this and the history of these and other groups are found in chapter 7.1.

What all of these attackers have in common is that they usually have access to the APK file, owning root rights on their phones, and therefore, they can access any part of a smartphone or intercept communications. For instance, in advance of cracking an app, it first needs to be analyzed and reengineered to allow for the desired modifications.

A reader might ask the question, why are there still so many issues on mobile platforms that seem to be solved on desktop computers already? This author assumes that one reason might be that Google chose to use Java language for their system because it is known to have these issues with easy decompilation possibilities, and they were able to solve a different issue that way, which Apple still faces today. Java is platform-independent and allows an unlimited amount of different devices, while there are only a few Apple smartphone devices available as of today. This might also be caused by the fact that Apple’s iOS is closed source and wants to remain the only reseller. Nevertheless, Google’s selection comes with some disadvantages that need to be solved separately now and sometimes it is possible to use well-known measurements from the desktop world on mobile computers to fight the issues as highlighted in this dissertation in the upcoming chapters.

[13] Infected app with malware

6.3 Problem statement

The main issues of copyright protection are shown in Figure 3 and can be summarized in one sentence by the fact that there are now software pirates circumventing copy protections that are mostly very weak, and the goal of this dissertation is to analyze and improve currently available methods for existing smartphones and tablet devices using Android. Additional details are outlined in the upcoming subsections.

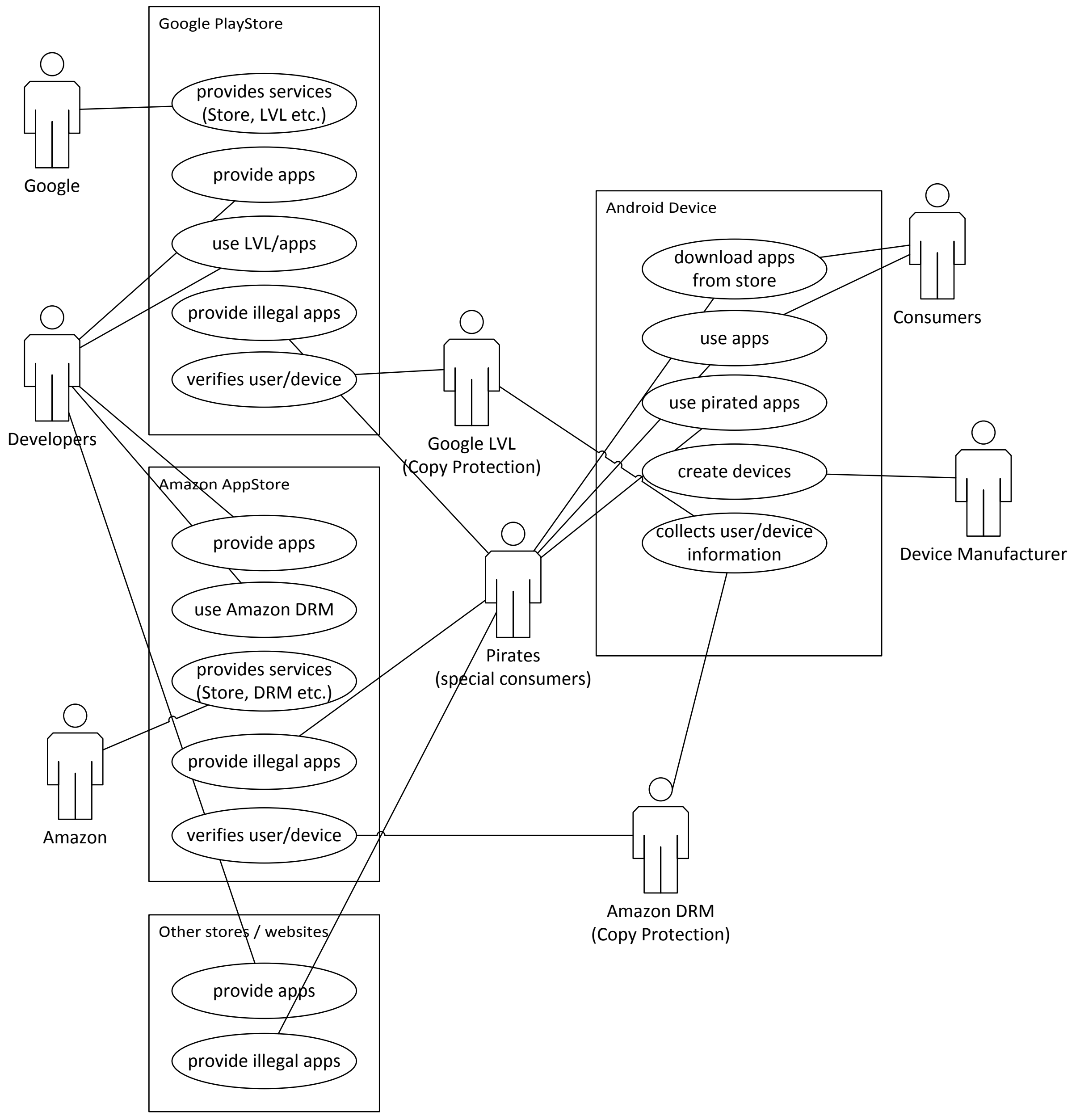


Figure 3 - Simplified situation overview (big picture of piracy- and copy protection issues)

6.3.1 Root access on Android devices

One of the key issues on current Android systems in terms of copy protection is the fact that devices may be rooted[14] either legally (option by the manufacturer, e.g. HTC [51]) or by using an exploit (see 10.1.2 for details). A rooted device permits its owner a modification of the system and access to all data privately stored by apps as well as data passing network and local connections. Therefore, any rooted device needs to be considered insecure in terms of copyright protection, since it may reveal the details of the mechanism to allow its circumvention. Nevertheless, in theory, for root users even the access to certain resources can be restricted with the introduction of Security-Enhanced Linux on Android (SEAndroid) with version 4.3 and finally, when it is enabled (enforced) in Android 5.0 [52]. For instance, this feature is used by Samsung KNOX, a security enhancement on Samsung devices, to ensure that only valid apps can access their data [53]. Enabling this feature in combination with other security measures like secure boot[15] and usage of TEE/SEs may tighten Android's security immensely and it ultimately relies on Google and the devices manufacturers to secure it [54].

Nevertheless, Android offers a sufficient attack surface that can still be rooted temporary at least (cf. exploits), while the bootloader may remain locked and it is up to its implementation how modifications on system partitions might be handled on a future reboot. For instance, this applies to some Verizon devices as stated by a user [55] on reddit and the phone can be rooted, while the bootloader stays locked. In fact, it would require severe interaction between many stakeholders to highly tighten the security of a device covering several existing issues in hardware and software. It must be assumed that any device may be rooted after a certain time due to upcoming exploits. Developing a copyright protection for rooted devices is not impossible, but it is not currently supported. Here, it would be preferred to have devices more secured and licensed data must be protected even from root access. Of course, "securing a device" could be understood in many different ways. In fact, there are devices available like the "Black" [56] by Boeing that provides highly sophisticated security measures against hardware manipulation or other tampering, and will render itself useless in case of any break-in attempts. Another meaning of securing a device may be related to data privacy and the protection of user data. In terms of this dissertation "securing a device" refers to hardening its copy protection.

Unfortunately, many device manufacturers like HTC [51], Motorola [57], Sony Ericsson [58], Samsung [59] and others allow the rooting of their devices, and permit customers to install a so-called custom rom[16] by unlocking the bootloader and flashing the desired data to partitions. Sometimes the manufacturers may even permanently flag a device as Samsung does [60]. In theory, Samsung's approach of blocking further KNOX container-usage, as stated in [60] might be an acceptable way for copyright-protection and the usage of certain apps may be prohibited in that case. Nevertheless, that technology is available on Samsung devices only, but it is questionable if customers will accept such intense limitations on purchased smartphone devices. Instead, on other devices like gaming consoles, it seems to be widely accepted already,

[14] rooted means the user and apps are able to acquire root rights on the underlying Linux system and control almost anything on the device.
[15] secure boot describes a secured way of booting by validating the signature of the loaded code before proceeding to boot it [54]
[16] Firmware by third parties, e.g., CyanogenMod

and there is most often no legal option to unlock a gaming console and modders[17] face the risk of getting their consoles banned permanently [61].

6.3.2 Reengineering of Android apps

Easy reengineering opportunities (cf. section 8.2ff for details) are another key issue on current Android systems. Exchanging and hiding licensing information or other confidential information is a tough challenge, but root access makes reengineering much easier, too. For example, the APK files and other internal files may be viewed (cf. ls –l /data/ not denied anymore) now, and some developers store secret codes within the shared preferences that are saved in the private app directories.

6.3.3 Interception of Android apps

A further key issue based on root-rights is the interception of any function calls or network traffic. For instance, it was used within our research to reengineer Java-based frameworks by Google (cf. section 11.4.8 about the nLVL as original outlined by [62]). The fact that all information may be intercepted as soon as an app (e.g., an attacker) runs with root privileges ((assuming access is not restricted for root users either (cf. even Android SE has exploits, too [63])), one can imagine the huge impact on hidden and protected sensitive information like license data, and encryption keys that may be revealed by tools like Frida[18] or the Xposed Framework[19] as already explained by [64, p. 54ff]. Moreover, it is covered in this dissertation in more detail in section 10.1.5.

6.3.4 Existing copyright protections

Android (Google) offers the License Verification Library (LVL) for app developers so far [47] to ensure that their apps have been bought and they own a valid license. Instead, Amazon automatically applies its “Amazon DRM” protection [46]. Alternative app markets like SlideMe provide further solutions, too [65]. Moreover, researchers provided some improved ideas (see 9.2ff), but there are no available figures about its usage and the mainstream apps are probably protected by Google’s or Amazon’s solutions only.

While Amazon’s solution may be circumvented easily (see section 10.1.4), it needs to be assumed that SlideMe’s SlideLock can be surrounded in a similar way like the LVL by modifying a single function (not practically verified). Other solutions do not seem to be available publically and therefore, are not analyzed. We also discovered that some developers implemented Google’s LVL library improperly (= without any changes). Examples are the “Worms” app by Team17, and we notified them as well (see 14.5). Using the default

[17] People who modify a firmware
[18] http://www.frida.re/
[19] http://repo.xposed.info

implementation allows software pirates an easy removal of that protection. For instance, by using method call interception as presented in detail in 10.1.5, which was requested by and based on [64, pp. 30ff, 54], or by using tools like “Lucky Patcher” [49] (details here in section 10.1.3 as originally outlined in and requested from [66]).

Also, it has to be assumed that many developers are unaware of the available reengineering tools and their easiness of use. Likewise developers are most often not educated about the issues, while performing app development in the suggested manner (cf. design patterns), which also makes it easier for reengineers. One certain issue is that even basic protection methods included with the Android SDK (e.g., ProGuard) were not enabled by Google for years by default. For instance, Gartner released a press release stating that “75 Percent [sic!] of Mobile Applications will Fail [sic!] Basic Security Tests” [67], which can be used as an indicator for the security knowledge of developers on copyright protection, too. As of today, it can be assumed that many apps use ProGuard’s optimizations due to Google’s recommendation on shrinking code [68]. This was backed at the recent Google IO as well [69]. Nevertheless, as recognized in our evaluations (see section 13ff), the obfuscation applied by ProGuard to our testing apps did not really stop attackers at all and it is simply too weak on the Java level. In fact, there are paid commercial tools (like DexGuard), but with the exception of its usage by financial institutes, there are no known statistics on its general distribution level. This was confirmed by Eric Lafortune[20] [35].

Nevertheless, even advanced developers cannot protect their software securely using the default developer tools provided by Google. They face similar issues with a time advantage (e.g., by using ProGuard) only. We analyzed this topic and freely available tools in more detail in several research works that are included in the “security analysis” (see chapter 10).

Tools by third parties, which were not available without costs (e.g. DexGuard) are observed, but not analyzed in detail and any information is based on provided, publicly available information.

6.3.5 Customers prefer free apps

As outlined in [70] most sales happen in the first 30 days of an app’s release, and the key is to find protection that protects the app that long. Naturally, customers prefer free apps against paid apps. While there are students, who would like to play a game, but do not have the money for it, it is common knowledge that in general, “Nobody likes to pay bills” [71]. A typical user does not feel bad about software piracy and most often will say common statements like “everybody does it”. In an evaluation by [72], this common assumption is confirmed: 68.3% of 640 students pirated software. Also, around 54% of both, pro-piracy and anti-piracy students, “believe that software is public property” [72, p. 73].

In general, there are at least two rivaling groups – on the one hand, the copyright holders claim it is theft, and other parties claim it has to be free as outlined in a report by [73]. For example, the lastter view is often found in research communities, where journals hold the copyrights, and scientists want everyone to have access to the information. In terms of Android apps, the author believes that it is simply theft, and companies developing these games sell the apps to gain

[20] Co-founder and CTO of GuardSquare ; producer of ProGuard/DexGuard

income. Others see the piracy related cracking just as a sport to crack the newest protection for fun (as outlined in 7.1 in more detail). Of course, some would like to try out games for free first, and the gaming industry satisfied customers' requests by adapting the model of freemium apps in recent years that offer additional services via in-app-billing [74]. For developers, it is important that both, copy protections as well as in-app-billing methods (see [75] for an attack example on in-app-billing), are prone to similar attack vectors and should be reviewed with the suggestions of this dissertation, since some of the ideas (e.g., porting to native code) can improve it as well.

6.3.6 Objectives and research questions

The main objective of this dissertation is to a identify better solutions for copyright protection on Android, while the proposed techniques may be of interest for related topics like data privacy and data protection in general.

We can specify the following research questions that are related to that goal:

No.	**Question**
0	Fundamental question: Are the current copyright protections for Android sufficiently secure?
1	If that is not the case, how can we ensure that an app is used on a valid device or by the valid user only?
2	Is it possible to store sensitive information like licensed data more securely, maybe, e.g., by using a secure element or alternatives?
3	Is it actually possible to use a secure element on Android (as a developer)?
4	How can we improve copyright protections and how can we implement them on Android?
5	How can we protect apps against reengineering (cf. static- and dynamic analysis) and is that actually possible with usual Android versions?
6	Might it be a better approach to use native code for security related issues instead of Java (cf. desktop world is dominated by native code and iOS uses it as well)?
7	What needs to happen elsewhere to improve the situation, (e.g., hardware modification and/or better cooperation by different manufacturers)?

6.4 Contribution summary

While it is recommended to read the full contribution/conclusion section at the end, a short summary should be presented in advance not requiring the details of all other chapters.

As outlined in 14.2, we were able to confirm earlier findings by others about the severe reengineering issues on Android in regard to its copyright protection (cf. License Verification Library (LVL) by Google) and even extend it to other security solutions like Amazon's DRM, while showing the proofs for the insecurity of Android in general.

In analyzing different options for gaining more security, the choice of using native code turned out to be the most effective one by comparing several examples of reengineered code and available methods for its protection (obfuscation). Ultimately, that led to the development of our proposed solutions of a native code version of the aforementioned LVL while researching additional methods - called fusing options - to bind native code (in our cases primarily used for licensing) and the app together to prevent not only their separation, but also attacks replacing function calls (or return values). In addition, methods for information exchange between different program parts (called indirect method triggering) allow more secured communication between different security functions across the app without revealing too much information to an attacker right away in comparison to usual function calls.

In addition, several conceptual solutions are presented using, e.g., Secure Elements (SEs). Furthermore, ideas that require the actions of stakeholders (like Google and device manufacturers) show and highlight even more secure solutions, which cannot be realized at this time.

Finally, we evaluated the security increase with different testing groups showing a significant improvement against the existing solutions used by major app markets.

6.5 Dissertation outline

Introduction (Section 6, including previous chapters)

The introduction chapter should give the reader a synopsis of the topic, providing a quick and summarized overview on fundamental knowledge for understanding the issues discussed in this dissertation. Furthermore, the motivation and actual problem are introduced, and the research questions of this work are highlighted.

General background (Section 7)

This section covers the fundamentals from an introduction to the history of software piracy to all important Android and hardware topics that should be known by a reader like, e.g., information on different Android versions, the development of apps, system internals and possible security solutions in hardware.

Topic-specific background (Section 8)

The main topic of reengineering is explored in high detail in Section 8. Here, typical reengineering tools are introduced. Moreover, information about possible, existing protections and attack vectors are presented and compared to those used in the desktop world.

Related work (Section 9)

This section presents an overview and short introduction on recent works in related areas as well as a comments about the relations to this work and/or issues with the proposed solutions. Any theses by our students are not included here, since they belong to our research group instead; Implementations and analyses were performed upon the author’s request and under his guidance.

Existing solutions and their challenges (Section 10)

The security analysis in this section highlights the current state of issues on Android security in general, along with all related issues in terms of copy protections by reviewing circumvention options using static or dynamic reengineering options. Furthermore, currently existing solutions by other vendors are elaborated and approaches such as using native code, in terms of Android are reviewed, while taking hardware protections options in mind as well.

Proposed Solution (Section 11)

While the security analysis offered insights on alternative options that provide more protection already, this section focuses on possible options to improve the current issues with copyright protection on Android. Besides reviewing the options for global players like Google or device manufacturers, several options for developers are introduced that allow a sufficiently secure implementation of copyright protections for Android than the existing solutions used by the major app markets.

Prototypic Implementation (Section 12)

While there are many options for creating a unique copyright protection based on the proposals of the previous section, this chapter introduces three possible implementations using different features that will be evaluated in the evaluation chapter.

Evaluation (Section 13)

The evaluation section reviews the implemented example apps, while also presenting the results from the performed evaluations in our Android practical course as well as any conclusions.

Summary (Section 14)

The summary section reviews the results of this work and highlights the contributions, while taking an outlook to future possibilities and open research issues.

Appendix (Section 15)

The appendix lists most sources codes that are referred in the text, while including proofs and other forms for information.

Further chapters

All further chapters are dedicated to abbreviations, lists and references.

7 General background

The following chapter should provide an introduction to fundamental Android topics such as its architecture, an overview to its versions, the general development of apps, its distribution channels, further details on the execution of these apps and their used runtime environment as well as details on files, important directories and other related topics that might be required in later chapters, while the reengineering of Android apps is covered in an own chapter due to its importance to this dissertation.

7.1 History – An introduction to the beginnings of software piracy

These (piracy) groups [76] have their origin in the so-called "The Scene" (also known as Warez Scene) that founded itself in the early 70-80s as a response to initial copyright protection mechanisms and consisted of hundreds of groups till the early 90s. They rivaled against each other for being the first in releasing an illegal copy. It is like a sport to them [77]. With the rise of the internet, they were able to initiate even more advanced structures to fulfill their goal of releasing any new software, movie, or music to the internet as soon as possible and also for free. After all of this, they drew the attention of the FBI and forced President Bill Clinton to sign the "No Electronic Theft Act"[21] [76] into law by 1997. That law enabled the FBI to file cases against piracy followed by razzias in 2001 ("Operation Buccaneer" [76]), as well as in Germany [77]. Feeling still powerless, the content industry started to notify usual customers about the issue and related penalties (e.g., the warnings before movies in theaters). Up until today, these groups continue their fight against the industry joined by researchers on one or the other side according to their personal interest. Of course, researchers should only follow legal paths and usually work closely with industry. While some of these release groups do it for fun and honor, others try to gain profit by placing commercials on their frequently visited websites [78] or use "'piggybacked' apps"[22] [48]. Nowadays, organized crime is also trying to gain profit with it [79]. It is an everlasting fight between the software industry and the crackers[23].

Over the years, people split up in different groups and according to their interest joined one side or the other; whichever propagated their way of thinking. For instance, the Free Software Foundation, Pirate Party, or Electronic Frontier Foundation should be mentioned because they engage themselves in supporting open-source and fair-use of software, as well as open-culture. On the other side organizations like the Business Software Alliance (BSA) or International Intellectual Property Alliance (IIPA) correspond with the interest of the copyright holders to claim their rights and criminalize anyone, who shares commercial software (or other data) but not in the same way they claim it (cf. complicated license agreements).

[21] Transcript from video by author
[22] Infected app with malware
[23] Person, who cracks software. The scene itself subdivides in different positions according to their tasks.

7.2 Definition of software, piracy and licensing

7.2.1 Software

Specifying the term software is not an easy task and there may be several definitions, e.g., as explained by [80] application software is the application everyone is using like Microsoft Word, while the underlying software is system software and refers to the operating system. Middleware describes software that acts in-between these layers like frameworks, also known from Android. In the end, software is a collection of processor instructions that are combined into an application binary file to be executed on the preferred platform/processor, e.g., x86 or the Dalvik VM.

7.2.2 Piracy

"Software piracy is the unauthorized copying, reproduction, use, or manufacture of software products" [81]. It may have its name origin from the real pirates that stole gold and other valuables in the Caribbean Sea around the 17th century. In addition to the provided definition by Microsoft, license violations such as using a purchased (or stolen) product on hundreds of computers while the license permits the installation only on a single computer, is also commonly known under the term software piracy.

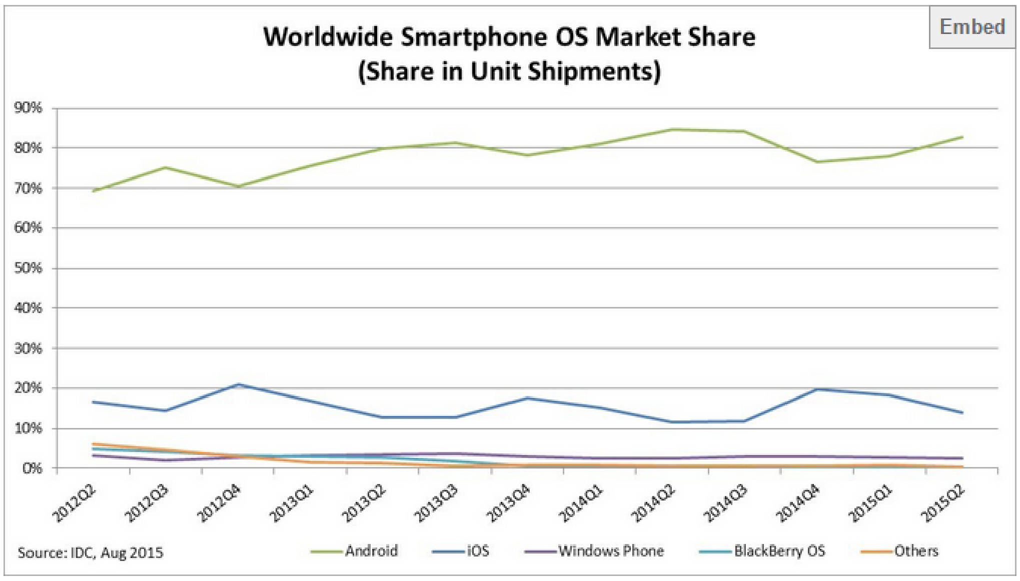
7.2.3 Licensing

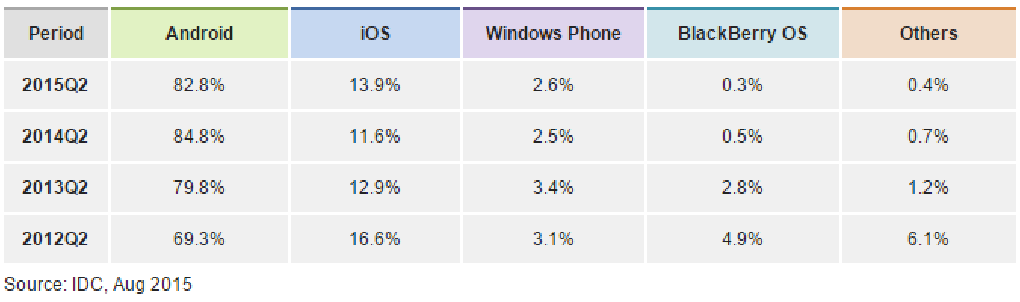
Software licensing describes a legal agreement between a customer and the copyright holder about "the legal rights pertaining to the authorized use of digital material" [82]. It can come in all variations from permitting customers to use an application on a single device by a single user or by the whole family or even for hundreds of computers. For example, many companies ask customers (in their software) to agree to a so-called EULA, the "End User License Agreement" [83] upon the installation of software. In addition, there are several open-source licenses, e.g., GPL, that permit, e.g., free and almost unlimited usage, in accordance with the license [84].

7.3 Android

Android is a mobile operating system by Google and the major target platform in this dissertation. Originally, Android Inc. was founded by Andy Rubin in 2003 before Google took it over in July 2005 [85]. Android is supported by the Open Handset Alliance - a group of companies, who are "committed to greater openness in the mobile ecosystem" [86]. Android is their first product and open-source [86].

Nowadays, its market share is enormous, and Android covered more than 80% of the market worldwide by 2015/Q2 [87], and with similar figures, e.g., 77% in Germany, in recent times (May 2016) [4]. Figure 4 illustrates its market share in comparison to other mobile systems on a timeline. One of the reasons for its success might be its openness and customization possibilities besides the fact that Google pushes it a lot as well. For instance, smaller start-up companies may use the system to turn it into a new product and one can find many examples across the internet and on platforms like Kickstarter[24]. Examples that can be mentioned here are the outdoor smartphones by TakWak Company or the modified Android versions like CyanogenMod by the company CyanogenMod LLC.





Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014Q2	84.8%	11.6%	2.5%	0.5%	0.7%
2013Q2	79.8%	12.9%	3.4%	2.8%	1.2%
2012Q2	69.3%	16.6%	3.1%	4.9%	6.1%

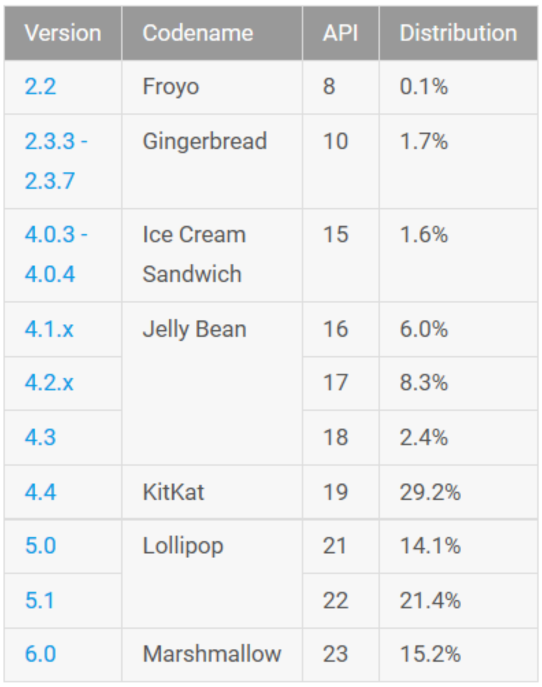
Source: IDC, Aug 2015

Figure 4 - Market shares [87]

7.3.1 Android versions

The most recent Android version released by Google is Android 6.0[25], codename Marshmallow and its updates, while "Nougat" [88] is still in preparation. Over time, Google (see Figure 7) has released several Android versions where (until today) each one features new functions and sometimes supports new hardware. Often Google releases a demo device that shows all new features. It is the Nexus series. Figure 5 shows the distribution of Android versions among registered Android devices as gathered by Google [89].

[24] Kickstarter.com is a platform for private individuals or smaller companies to raise money to cover initial development costs etc. ; Anyone can make a pledge and support the presented ideas
[25] according to Android.com by July 13th 2016



Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.6%
4.1.x	Jelly Bean	16	6.0%
4.2.x		17	8.3%
4.3		18	2.4%
4.4	KitKat	19	29.2%
5.0	Lollipop	21	14.1%
5.1		22	21.4%
6.0	Marshmallow	23	15.2%

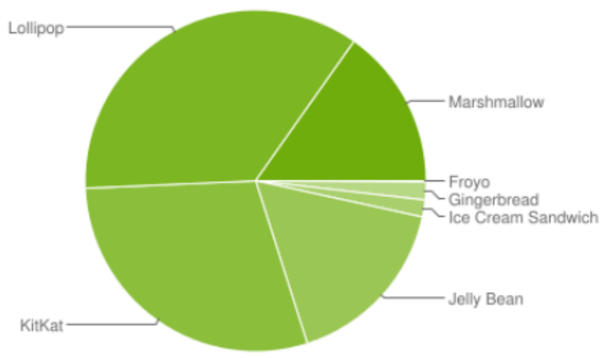
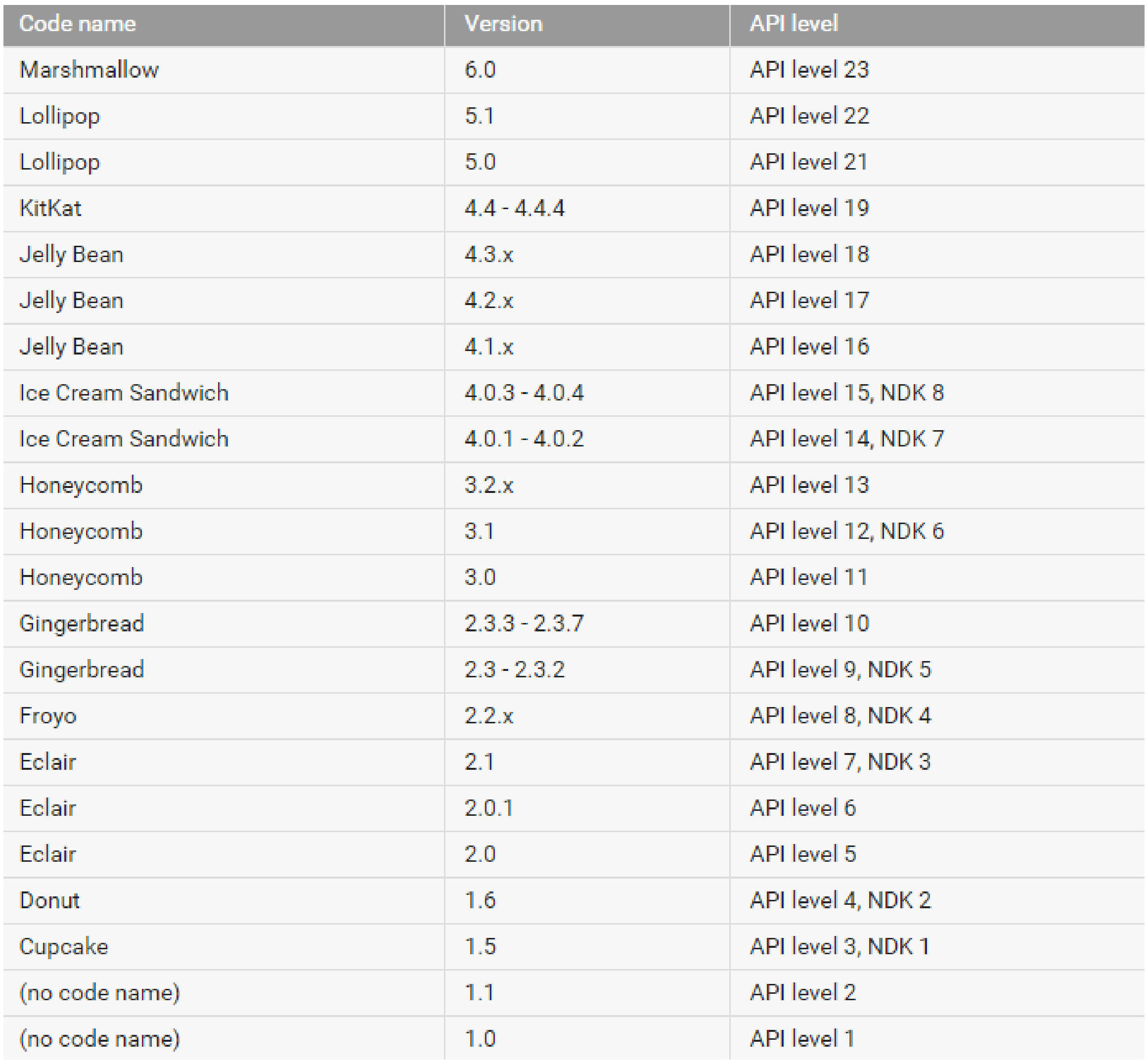


Figure 5 - Different Android versions among the Google ecosystem collected by August 1st 2016 [89]

Depending on the version (see Figure 6 and Figure 7), Android offers a different feature set (API/NDK version). For instance, to use external SEs (Secure Elements) USB-OTG is required and it was introduced to Android in 2011 with the release of Android 3.1 aka Honeycomb (see Figure 7) for tablet devices [90]. Moreover, different branches for smartphones and tablets were resolved in 2011 with the release of Android's 'Ice Cream Sandwich' (see Figure 7). In addition to the official Android versions, the Internet community worked on several modifications of Android and released them under different names, each with a different feature set. Some of these modifications (commonly known as mods) focus on privacy related issues, while others try to improve the performance by adding additional features (e.g., root rights) in general. Famous examples that can be mentioned might be CyanogenMod and MIUI [91] [92].



Code name	Version	API level
Marshmallow	6.0	API level 23
Lollipop	5.1	API level 22
Lollipop	5.0	API level 21
KitKat	4.4 - 4.4.4	API level 19
Jelly Bean	4.3.x	API level 18
Jelly Bean	4.2.x	API level 17
Jelly Bean	4.1.x	API level 16
Ice Cream Sandwich	4.0.3 - 4.0.4	API level 15, NDK 8
Ice Cream Sandwich	4.0.1 - 4.0.2	API level 14, NDK 7
Honeycomb	3.2.x	API level 13
Honeycomb	3.1	API level 12, NDK 6
Honeycomb	3.0	API level 11
Gingerbread	2.3.3 - 2.3.7	API level 10
Gingerbread	2.3 - 2.3.2	API level 9, NDK 5
Froyo	2.2.x	API level 8, NDK 4
Eclair	2.1	API level 7, NDK 3
Eclair	2.0.1	API level 6
Eclair	2.0	API level 5
Donut	1.6	API level 4, NDK 2
Cupcake	1.5	API level 3, NDK 1
(no code name)	1.1	API level 2
(no code name)	1.0	API level 1

Figure 6 - Android versions, their names and related API levels [93]

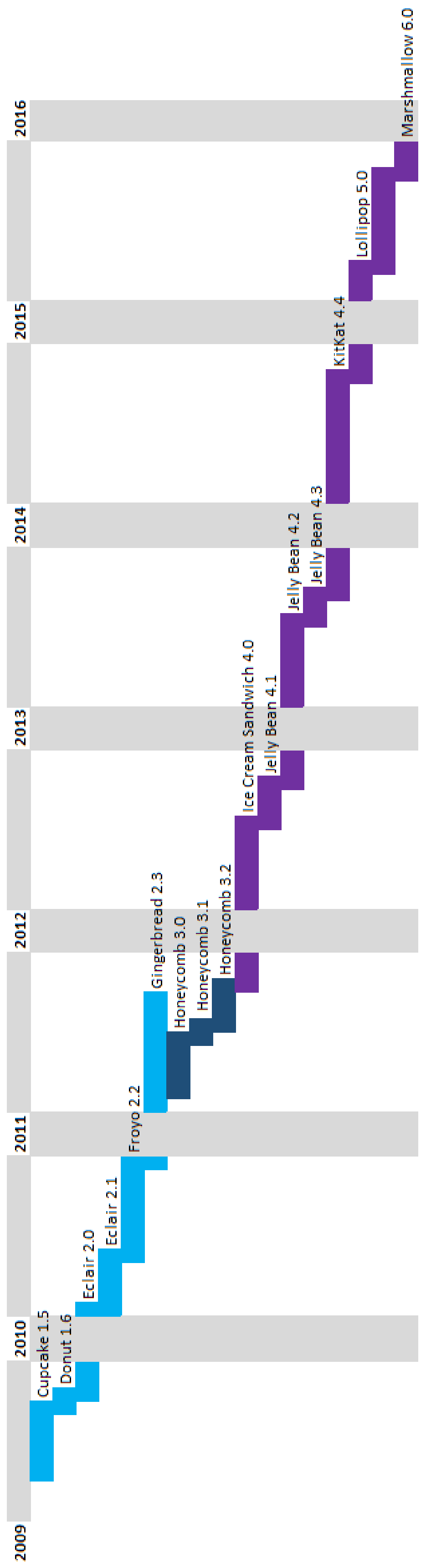


Figure 7 - Android versions on a timeline (based on [94])

7.3.2 Android architecture

Android was originally optimized for low-performance devices and was mainly intended to work on ARM CPUs [95]. Nevertheless, new Android versions such as Lollipop support x86- and also MIPS-based devices [96].

Android and its relation to Linux

Figure 8 illustrates Android's general architecture as presented by Bornstein in the beginnings. It shows that Android is based on Linux and many open-source libraries. On top of everything the Android applications run by using their required frameworks and then they are executed by the Dalvik or — nowadays — ART VM. Figure 9 reveals further details and a comparison to Linux itself. For instance, Android uses a reduced glibc library (called bionic) in comparison to Linux and Android apps are using the desired frameworks (see Figure 8) to access the hardware, while running in a virtual machine [97].

Thinking about the system startup [98, p. 809f], the first process initiated by the kernel is 'init' as in a traditional Linux system. It is the root of all further processes. It starts daemons that are focused on low-level tasks, e.g., adbd and Zygote. Zygote itself starts all "higher-level Java language processes" [98, p. 809] like the "system_server" [98, p. 809] daemon responsible for the core services including the "power manager, package manager, window manager, and activity manager" [98, p. 810].

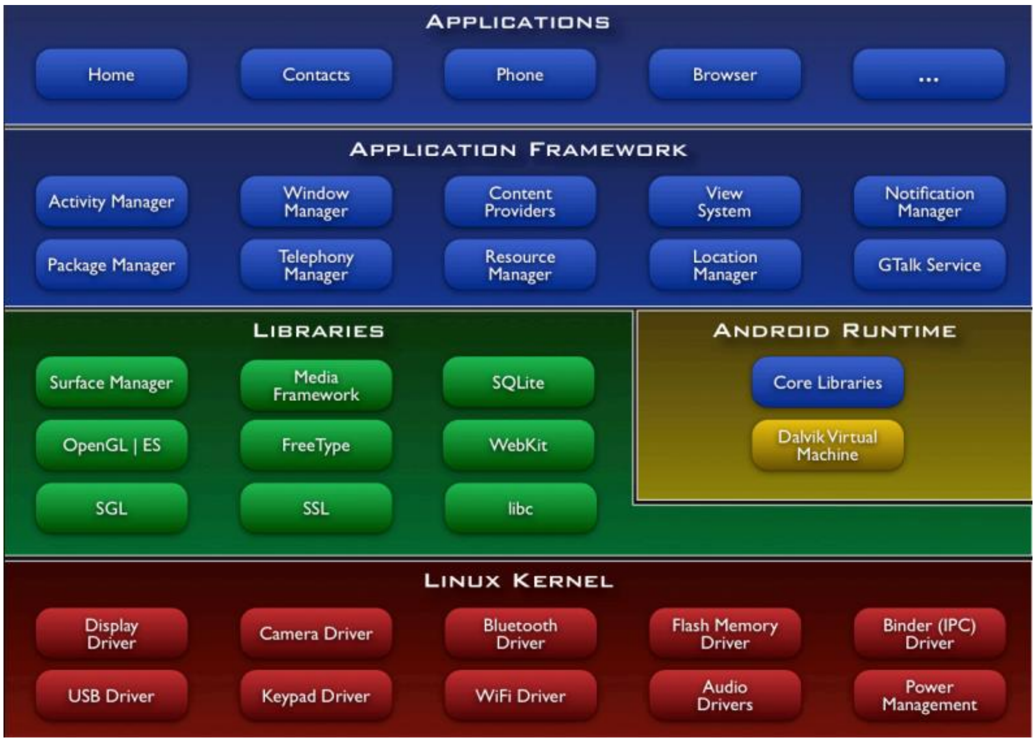


Figure 8 - Android Architecture [95]

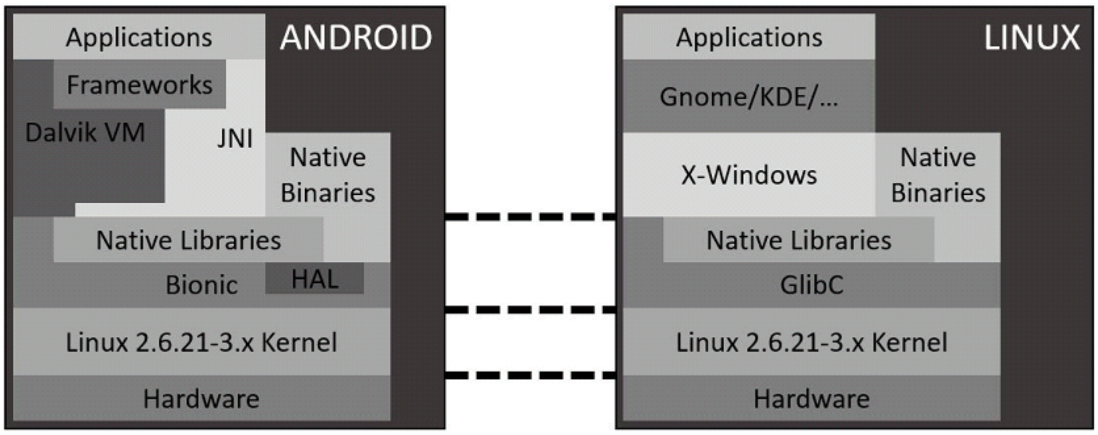


Figure 9 - Android Architecture vs. Linux Architecture [[99] as quoted in [100, p. 5]]

Differences to typical Linux computer systems (examples)

While Android is based largely on a stock Linux kernel, it provides different features sometimes. For instance, a feature special to mobiles is the wake lock option providing different power states to keep the hardware partially turned on to react to incoming calls or even cloud messaging [98, p. 810ff].

Further differences are the way it handles low memory situation. The implementations on Android act much more aggressively and Android uses no swap space. Therefore, low-memory situation are more common than on desktop computers. On typical desktop systems the implementation acts more like an emergency procedure killing the largest process, while honoring some other properties, too [98, p. 813]. Instead, "Android's out-of-memory killer uses […] parameter oom_adj [for scoring] […], but with strict ordering: processes with higher oom_adj will be killed before those with lower ones" [98, p. 814]. The file /proc/pid/oom_adj is used by Linux for scoring purposes to make a decision on which process should be killed in low memory situations [101].

Security architecture

Android makes use of Linux's user and group model for defining access rights to certain resources and apps as well as corresponding app directories get a user and group id [102].

With the release of Android L, Google activated another security feature called "SELinux" [52] (also called "SEAndroid" [103]) to intensify the access control even further and block the usage of resources (e.g., access to device files) for unprivileged apps and services.

Furthermore, Android's general security architecture and therefore, the access to the hardware or services is regulated on Android with regard to apps using predefined permissions, e.g., "android.permission.RECEIVE_SMS" [102] to receive text messages. Each developer is required to define them in the Manifest file[26] of an app, while the user has to accept all required permissions upon the installation of an app. Newer Android versions (>6.0) support permission granting on demand, too [102].

[26] Basic app configuration file containing name, version, required permissions etc.

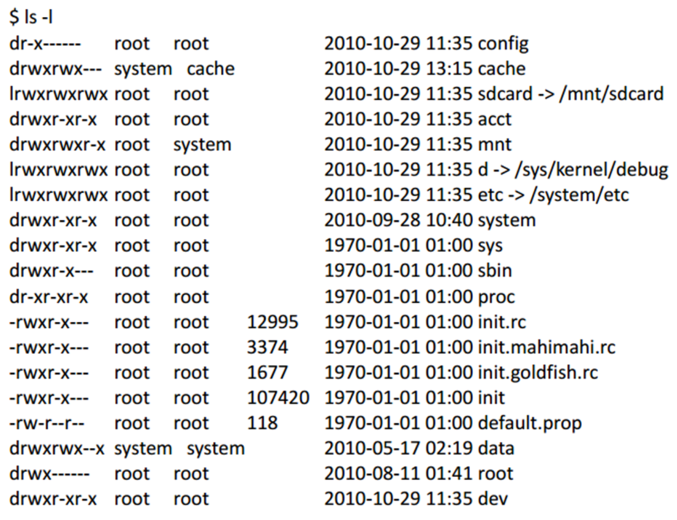
Partitions, filesystems and access rights

Moreover, Android has several partitions [104] for different purposes that may be gathered by executing “cat /proc/mtd” [104] on a device within a terminal application. While the misc-partition may be used for device-specific configurations, the boot- and recovery-partitions are both providing a “kernel […] [and] initrd with rootfs” [104]. The system-partition includes the actual Android operating system, while all data by the users are stored on the userdata-partition. Temporary data and related files are instead stored on the cache-partition [104].

The partitions, namely system, cache, and userdata [104] are mounted within Android’s filesystem at /system, /cache, and /data (same order). While the boot-partition always belongs to the corresponding Android system, the recovery-partition may be flashed with an alternative recovery software (e.g., ClockworkMod[27] or TWRP[28]) for increased features such as backing up the whole device. One of the default tools within the Android SDK for this purpose is fastboot. Originally Android used YAFFS2 as its filesystem, but changed it to EXT4 by 2010 to avoid (“bottleneck” [105]) issues with upcoming multicore CPUs [105].

By default, users do not have full access to the partitions with the exception of the (virtual[29]) SDcard directory that is mounted within the userdata partition for reading and writing. Nevertheless, this limitation may be removed on rooted devices, which is also a critical issue in terms of data and software security. Details are addressed in 10.1.2.

A view at a listing of Android’s root filesystem (see Figure 10) reveals its close relation to Linux and many directories (e.g., /dev/, /root/, etc.) may also be found on a usual Linux system. One important fact to mention is that most files and directories belong to the system or root user and are mostly not readable by the actual device-user. For instance, system permissions are required for upgrading the system as well as viewing the private data files of an application. Again, this limitation may be circumvented on rooted devices.



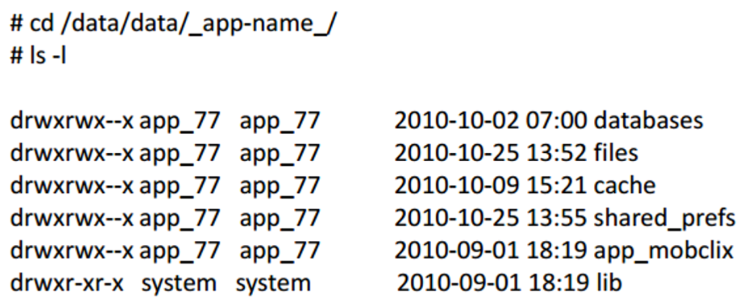
```
$ ls -l
dr-x------  root    root           2010-10-29 11:35 config
drwxrwx---  system  cache          2010-10-29 13:15 cache
lrwxrwxrwx  root    root           2010-10-29 11:35 sdcard -> /mnt/sdcard
drwxr-xr-x  root    root           2010-10-29 11:35 acct
drwxrwxr-x  root    system         2010-10-29 11:35 mnt
lrwxrwxrwx  root    root           2010-10-29 11:35 d -> /sys/kernel/debug
lrwxrwxrwx  root    root           2010-10-29 11:35 etc -> /system/etc
drwxr-xr-x  root    root           2010-09-28 10:40 system
drwxr-xr-x  root    root           1970-01-01 01:00 sys
drwxr-x---  root    root           1970-01-01 01:00 sbin
dr-xr-xr-x  root    root           1970-01-01 01:00 proc
-rwxr-x---  root    root    12995  1970-01-01 01:00 init.rc
-rwxr-x---  root    root    3374   1970-01-01 01:00 init.mahimahi.rc
-rwxr-x---  root    root    1677   1970-01-01 01:00 init.goldfish.rc
-rwxr-x---  root    root    107420 1970-01-01 01:00 init
-rw-r--r--  root    root    118    1970-01-01 01:00 default.prop
drwxrwx--x  system  system         2010-05-17 02:19 data
drwx------  root    root           2010-08-11 01:41 root
drwxr-xr-x  root    root           2010-10-29 11:35 dev
```

Figure 10 - Root filesystem of Android [106]

[27] https://www.clockworkmod.com/
[28] https://twrp.me/
[29] nowadays mostly emulated and many phones do not provide a SDcard slot anymore

Storage of apps and related security facts

By default, apps are stored as APK-files in the directory /data/app (see Figure 10) and all its corresponding data in /data/data/_app-name_. Here (see Figure 11), possible files and directories are the databases by an app as well as any created, private files. The owner and group of that directory will always be the app itself (cf. user- and group id) and it can manage the access correspondingly with an exception for the native libraries that the system owns. One of the key security features of Android [102] is to use Linux's user and group model to manage the access to resources. Each application is sandboxed that way and cannot access other apps' files. On newer Android versions there are additional security measures integrated, such as SE Android that enforces mandatory access control, which may limit the possibilities of a root user and ultimately, the effect of a compromised device to its owner [52]. As outlined in 11.4.7, there not only are advantages to this, but it may even lower security by preventing access to secured external devices (e.g., Secure Elements (SEs)).



```
# cd /data/data/_app-name_/
# ls -l

drwxrwx--x app_77   app_77        2010-10-02 07:00 databases
drwxrwx--x app_77   app_77        2010-10-25 13:52 files
drwxrwx--x app_77   app_77        2010-10-09 15:21 cache
drwxrwx--x app_77   app_77        2010-10-25 13:55 shared_prefs
drwxrwx--x app_77   app_77        2010-09-01 18:19 app_mobclix
drwxr-xr-x  system   system        2010-09-01 18:19 lib
```

Figure 11 - Directory of an Android App (example) [106]

In general, Android improves embedded binary files in APK files and stores optimized versions of apps in /data/dalvik-cache [64, p. 62]. These optimized versions are called ODEX files ("Optimized Dalvik Executable" [75, p. 23]) and stored in a file following the naming convention "data@app@PACKAGE_NAME.apk@classes.dex" [75, p. 23].

Instead, on newer Android versions (see 7.3.9) the OAT file is stored as "data@app@PACKAGE_NAME.apk@base.apk@classes.dex" [75, p. 27] and the folder /data/dalvik-cache/profiles instead. It is important to know that it is not a DEX file anymore as even the name surrogates something else [75, p. 27].

7.3.3 Android app development in general

Java by default (Android SDK) is used to develop applications for Android. In addition, it is possible to use C/C++ (Android NDK) [107].

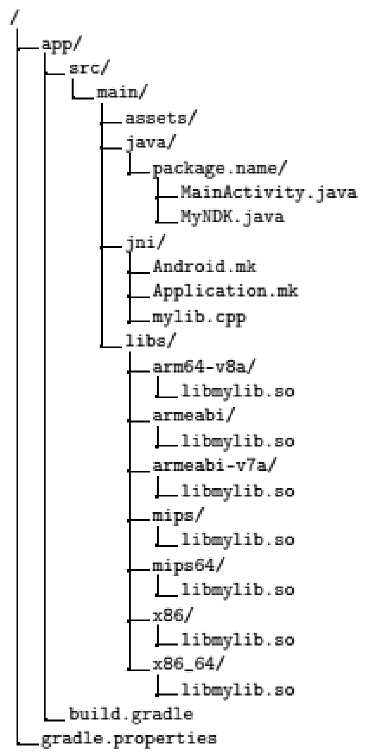
There are different Integrated Development Environments (IDEs) available today. On the one hand the first tool available - also used by the author - was Eclipse with additional Android-related plugins to allow the editing of layouts and the actual compilation process.

By the end of 2014 [108] Google upgraded Android Studio from its beta stage to its official Android IDE and version '1.0'. For instance, this IDE offers improved layout editors with preview options for different screens and hardware types. Moreover, the structure of projects is slightly modified and other configuration files are required due to the usage of gradle[30] for building apps.

In addition to the official IDEs, it is possible to use and develop apps using third party tools. Famous tools to be mentioned might be Xamarin [64] that allows cross-platform development of apps as well as Codename One [109]. For platform-independent convenience, there are also cloud-based IDEs (e.g., Codeenvy[31]) that allow the whole development process from any browser on any system. Also, for beginners, Google worked within its Google Labs division on a tool named “App Inventor” [110]. Nowadays, it’s still supported and improved by MIT [110].

7.3.4 Native app development

While Android apps are usually developed using Java, Google provides app developers the option to integrate native code (C/C++ code) into their apps by using the Android NDK [7]. Depending on the IDE used, the approach to use it for development is slightly different. A full guide for Android Studio - the current official IDE for Android - may be obtained from [7] as well as in illustrated form in [111].



```
/
└─app/
  └─src/
  │ └─main/
  │   ├─assets/
  │   ├─java/
  │   │ └─package.name/
  │   │   ├─MainActivity.java
  │   │   └─MyNDK.java
  │   ├─jni/
  │   │ ├─Android.mk
  │   │ ├─Application.mk
  │   │ └─mylib.cpp
  │   └─libs/
  │     ├─arm64-v8a/
  │     │ └─libmylib.so
  │     ├─armeabi/
  │     │ └─libmylib.so
  │     ├─armeabi-v7a/
  │     │ └─libmylib.so
  │     ├─mips/
  │     │ └─libmylib.so
  │     ├─mips64/
  │     │ └─libmylib.so
  │     ├─x86/
  │     │ └─libmylib.so
  │     └─x86_64/
  │       └─libmylib.so
  └─build.gradle
└─gradle.properties
```

Figure 12 - Android project with native Code (Android Studio) [100, p. Appendix A]

By default, as explained in [100, p. 33ff] (based on [7] and [111]), an Android project with integrated native code is structured as shown in Figure 12. Here the jni-folder contains the important native code- and make-files. The libs-folder includes the compiled versions of that code (shared libraries / *.so) for different platforms, e.g., x86 and armeabi. Related example

[30] Open-Source build system- https://gradle.org/
[31] https://codenvy.com

source codes may be found in the Appendix (see 15.1.2). Information on the actual build process may be found in 7.3.11.

In general, Android uses JNI, the “Java™ Native Interface” [112] like Java itself, too. It is the standard interface to provide native methods and use Java functions from native code [112].

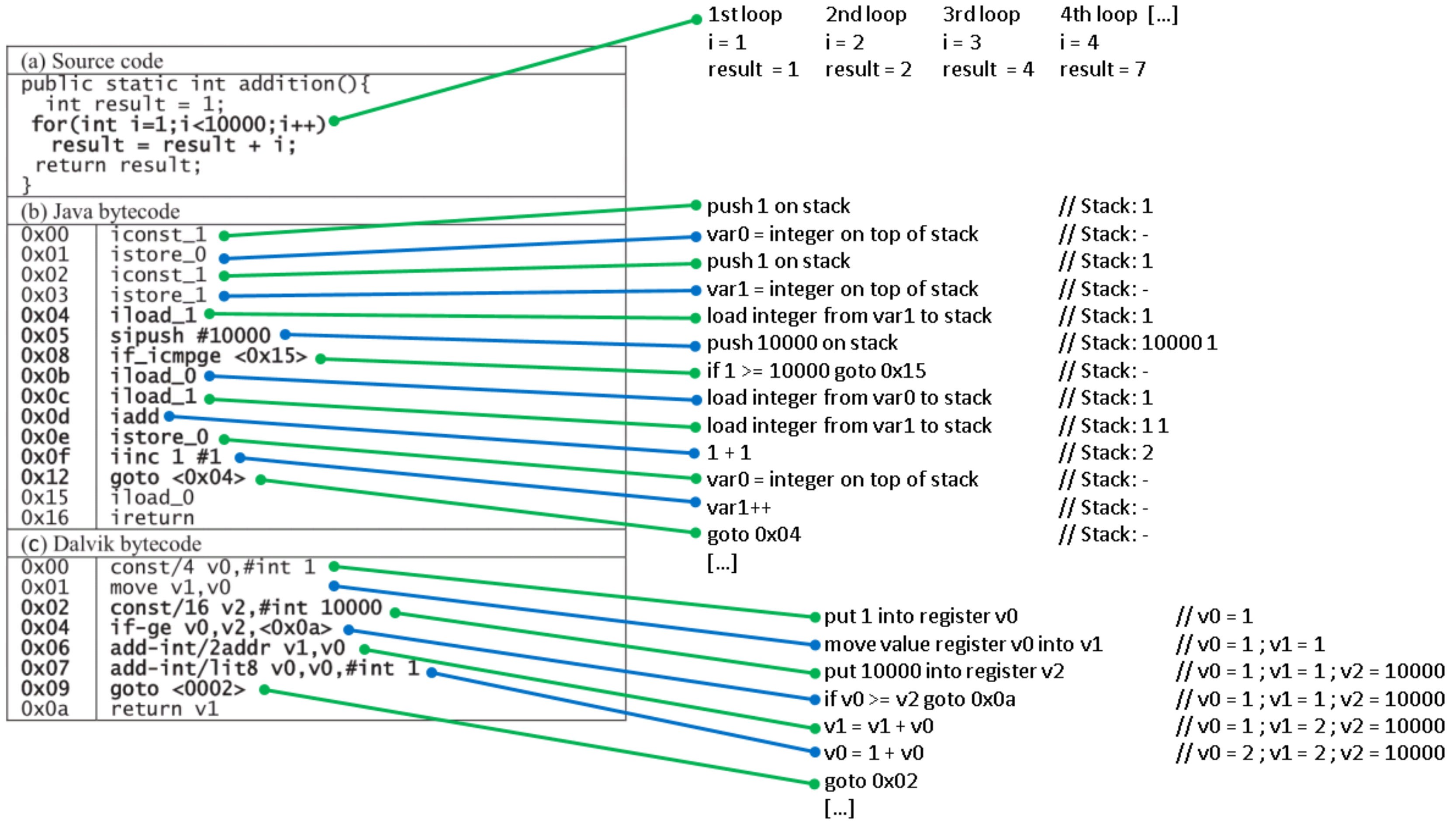
7.3.5 App distribution channels

At the beginning Google’s Play Store (initially named app market) provided the only access to apps (excluding the user’s option to install APK files). Over time, additional companies and other developers discovered the idea of offering alternatives for customers to receive apps. Reasons for this might be to have a more favorable revenue share, better content control, or less restrictions compared to Google’s Play Store, besides offering improved focus on different topics (e.g., open-source apps only as addressed by FDroid[32]). One of the larger competitors is probably Amazon with its Amazon AppStore. Others might be Wandoujia and AppChina known mainly on the Chinese market. Further options are GetJar, the Opera Mobile Store, and SlideMe as well as many others. Despite publishing apps on each app market, developers may use the services of publishing companies like CodeNgo that submits an app to multiple app stores programmatically [113].

7.3.6 The Dalvik VM

The Dalvik Virtual Machine [95] (DVM) and its name refers to a town in Iceland and is related to its creator Dan Bornstein. It is a register-based machine in contrast to usual stack-based CPUs or other virtual machines. For instance, this was done to “avoid instruction dispatch […] [and] unnecessary memory access” [95]. Figure 13 shows a comparison highlighting a much smaller code size for the same source code. An explanation on the used OPcodes in these figures is provided inline. The full references (see [114] [115] [116]) are of importance for understanding larger examples only. Originally, the Dalvik VM was optimized for slower CPUs with minimal RAM and acts gently on resources, while also being powered by battery [95]. Nowadays, some of these original goals are not valid anymore and current smartphones offer plenty of space, memory, and CPU power, while battery-life remains an issue. Basically, the “DVM is a customized and optimized version of the Java Virtual Machine (JVM) […] [and] Even [sic!] 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” [66, p. 15] (based on [117] and [118]).

[32] https://f-droid.org/



(a) Source code

```
public static int addition(){
  int result = 1;
 for(int i=1;i<10000;i++)
   result = result + i;
 return result;
}
```

```
1st loop        2nd loop        3rd loop        4th loop  [...]
i = 1           i = 2           i = 3           i = 4
result  = 1     result = 2      result  = 4     result = 7
```

(b) Java bytecode

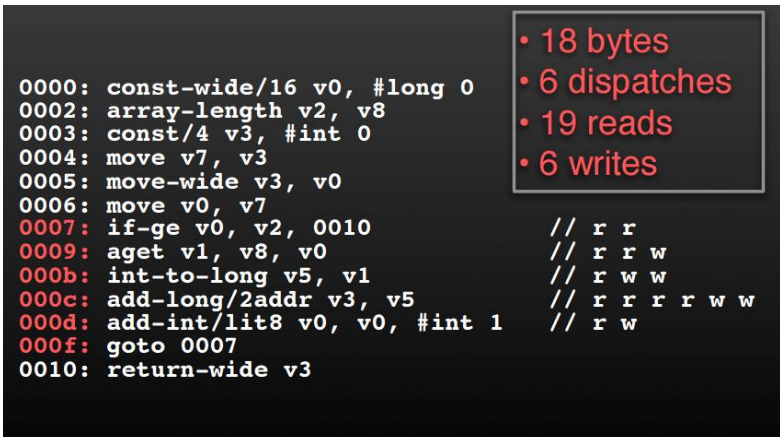
```
0x00   iconst_1            push 1 on stack                       // Stack: 1
0x01   istore_0            var0 = integer on top of stack        // Stack: -
0x02   iconst_1            push 1 on stack                       // Stack: 1
0x03   istore_1            var1 = integer on top of stack        // Stack: -
0x04   iload_1             load integer from var1 to stack       // Stack: 1
0x05   sipush #10000       push 10000 on stack                   // Stack: 10000 1
0x08   if_icmpge <0x15>    if 1 >= 10000 goto 0x15               // Stack: -
0x0b   iload_0             load integer from var0 to stack       // Stack: 1
0x0c   iload_1             load integer from var1 to stack       // Stack: 1 1
0x0d   iadd                1 + 1                                 // Stack: 2
0x0e   istore_0            var0 = integer on top of stack        // Stack: -
0x0f   iinc 1 #1           var1++                                // Stack: -
0x12   goto <0x04>         goto 0x04                             // Stack: -
0x15   iload_0             [...]
0x16   ireturn
```

(c) Dalvik bytecode

```
0x00   const/4 v0,#int 1            put 1 into register v0                // v0 = 1
0x01   move v1,v0                   move value register v0 into v1        // v0 = 1 ; v1 = 1
0x02   const/16 v2,#int 10000       put 10000 into register v2            // v0 = 1 ; v1 = 1 ; v2 = 10000
0x04   if-ge v0,v2,<0x0a>           if v0 >= v2 goto 0x0a                 // v0 = 1 ; v1 = 1 ; v2 = 10000
0x06   add-int/2addr v1,v0          v1 = v1 + v0                          // v0 = 1 ; v1 = 2 ; v2 = 10000
0x07   add-int/lit8 v0,v0,#int 1    v0 = 1 + v0                           // v0 = 2 ; v1 = 2 ; v2 = 10000
0x09   goto <0002>                  goto 0x02
0x0a   return v1                    [...]
```

Figure 13 - Comparison of bytecode in Java (b) and DEX (c) files (based on [64, p. 18] [114] [115] [116])

Another example (see Figure 14) taken from Dan Bornstein's slides [95] reveals optimized dispatches, writings as well as a smaller code size, while increasing the reading of certain registers, even more.



```
0000: const-wide/16 v0, #long 0
0002: array-length v2, v8
0003: const/4 v3, #int 0
0004: move v7, v3
0005: move-wide v3, v0
0006: move v0, v7
0007: if-ge v0, v2, 0010                // r r
0009: aget v1, v8, v0                   // r r w
000b: int-to-long v5, v1                // r w w
000c: add-long/2addr v3, v5             // r r r r w w
000d: add-int/lit8 v0, v0, #int 1       // r w
000f: goto 0007
0010: return-wide v3
```

- 18 bytes
- 6 dispatches
- 19 reads
- 6 writes

Figure 14 - Assembler of a DEX file [95, p. 40]

The compilation process for this VM is quite similar to the one for Java applications, but in an additional step the created JAR file with its CLASS files is converted into a special file-format called the DEX file that is optimized for embedded devices, differently structured and often smaller than the original JAR file. This conversion is called "cross-compilation", since the target platform (ARM) is other than the local one (e.g., x86). Figure 15 illustrates this conversion process and reveals, e.g., a shared constant pool as one of the differences [95].

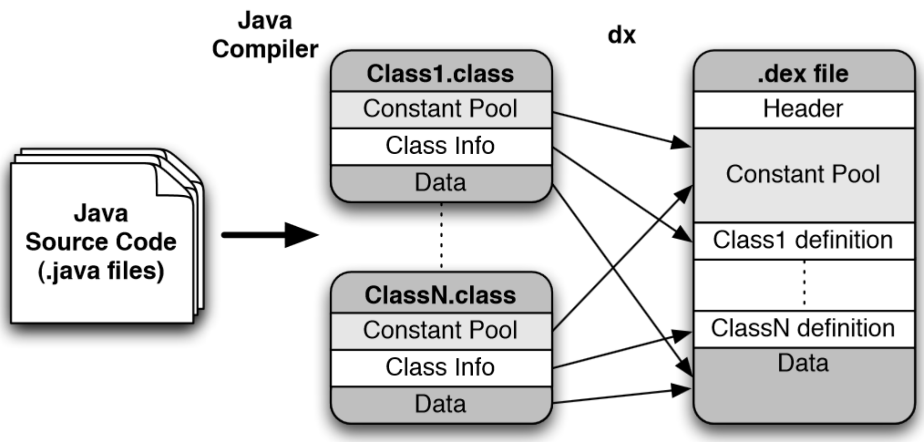
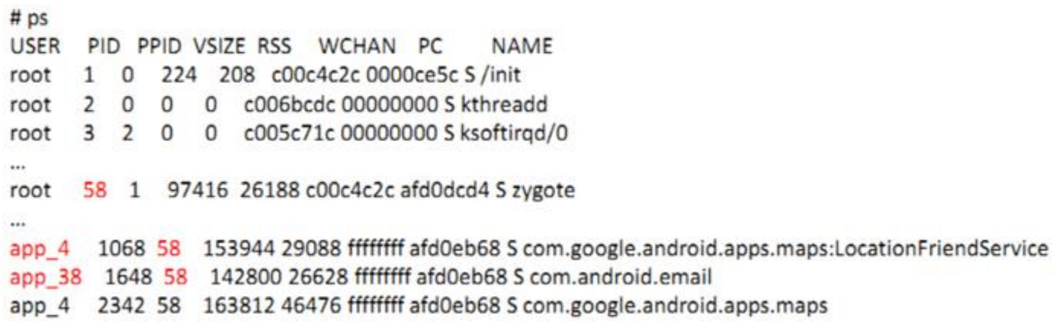


Figure 15 - DEX Conversion [119]

7.3.7 Zygote

On Android itself, the applications get executed by a parent process called "Zygote" that shares core libraries with its children (all apps) to once again save some memory (see Figure 16) [95] and for speeding up the app start significantly [100, p. 10] (based on [120]). The figure also highlights the different user ids for each app that represent one of the key security features of Android, since each app has its own id (and therefore, limited access rights) as seen below.



```
# ps
USER     PID  PPID  VSIZE  RSS    WCHAN     PC        NAME
root     1    0     224    208    c00c4c2c 0000ce5c S /init
root     2    0     0      0      c006bcdc 00000000 S kthreadd
root     3    2     0      0      c005c71c 00000000 S ksoftirqd/0
...
root     58   1     97416  26188 c00c4c2c afd0dcd4 S zygote
...
app_4    1068 58    153944 29088 ffffffff afd0eb68 S com.google.android.apps.maps:LocationFriendService
app_38   1648 58    142800 26628 ffffffff afd0eb68 S com.android.email
app_4    2342 58    163812 46476 ffffffff afd0eb68 S com.google.android.apps.maps
```

Figure 16 - Zygote and its child processes (based on [106])

7.3.8 The ART VM

In recent years the way of executing applications changed and first Google introduced its JIT (Just-In-Time) compilation [121] for the Dalvik VM followed by its OAT (Ahead-Of-Time) compilation for the new virtual machine called ART VM [122]. The main difference is the pre-compilation of DEX-files to native code, which slows down the installation process (and sometimes the system updates due to renewed optimizations), but increases the speed upon execution besides other benefits for battery life (e.g., no more wasteful JIT), multitasking, and - for future purposes - 64-bit support [122]. Figure 17 shows the differences in a diagram, when handling the APK and its embedded DEX-file.

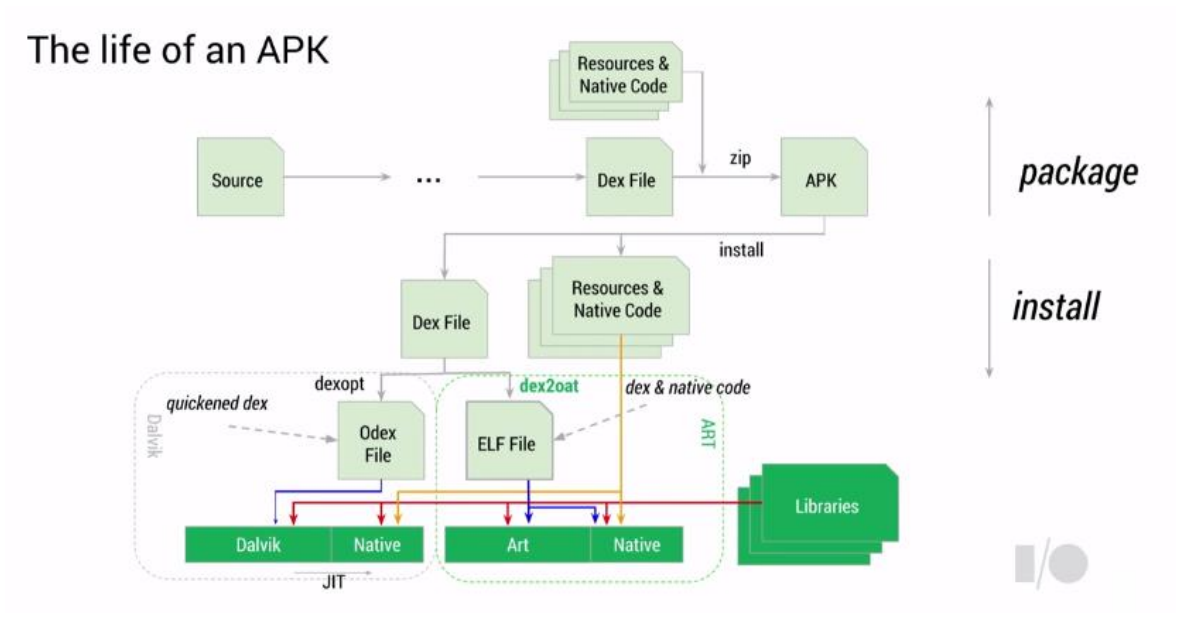


Figure 17 - Dalvik vs. ART VM [122]

The ART VM uses several files for execution including boot.art, boot.oat and (referring to any app) the ODEX file(s) (in OAT format now) [117, p. 11]. Further details are available in the next section.

In fact, the pre-compiled files represent an ELF file as "specified by UNIX System Laboratories (USL) and later by Tool Interface Standards (TIS) and is a common standard for executables, object code and shared libraries on UNIX [Linux] systems" [100, p. 13].

As mentioned in [100, p. 13] it is important to highlight that these files are shared object files and cannot be directly executed. In addition [100, p. 15] concludes that Android ELF files contain considerably less sections and segments than usual programs with over 30 estimated sections.

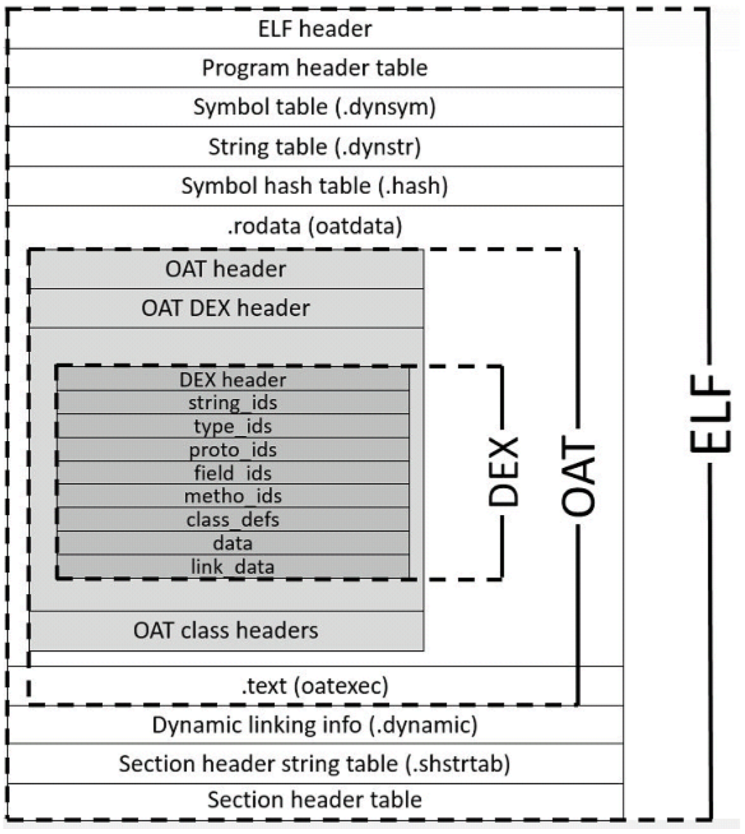


Figure 18 - ART Executable [100, p. 18] (based on [123] [124])

Figure 18 illustrates the structure of such a file. As explained by [100, p. 18] (based on [123] [124])

- the **ELF header** starting “at address 0x00 […] contain[s] information about the version, file type, target machine and offsets to the program- and section header tables” [100, p. 13] (based on [123])
- the **program header table** “is an array of structures, each describing a segment or other information the system needs to prepare the program for execution” [125]
- the **symbol table (.dynsym)** provides “information for locating and relocating a program’s symbol definitions and references” [100, p. 13] (based on [123]), e.g., oatexec
- the **string table (.dynstr)** includes strings
- the **symbol hash table (.hash)** provides the symbol hash table
- the **.rodata (oatdata) section** may contain any data and stores the OAT files with its embedded DEX files (see separated section about OAT and DEX files below)
- the **.text (oatexec) section** holds the program code
- the **.dynamic linking info (.dynamic)** “includes dynamic linking information” [100, p. 13] (based on [123])
- the **section header string table (.shstrtab)** includes the section names

- the **section header table** is an array of structures that allows the location of all file sections [125]

7.3.9 APK, DEX, ODEX and ART, OAT format

Android Application Packages (APK files)

APK files [100] contain resources and executable codes and are the default shipping format for Android applications. A standard APK file may look like what is shown in Figure 19 and includes an AndroidManifest.xml file with basic settings, the actual executable code in classes.dex, resources (layouts, images etc.) within the res folder, native libraries in the lib folder, certificates/signatures in the meta-inf folder, as well as more information about the actual resources in the previously mentioned folder in the file resources.arsc.

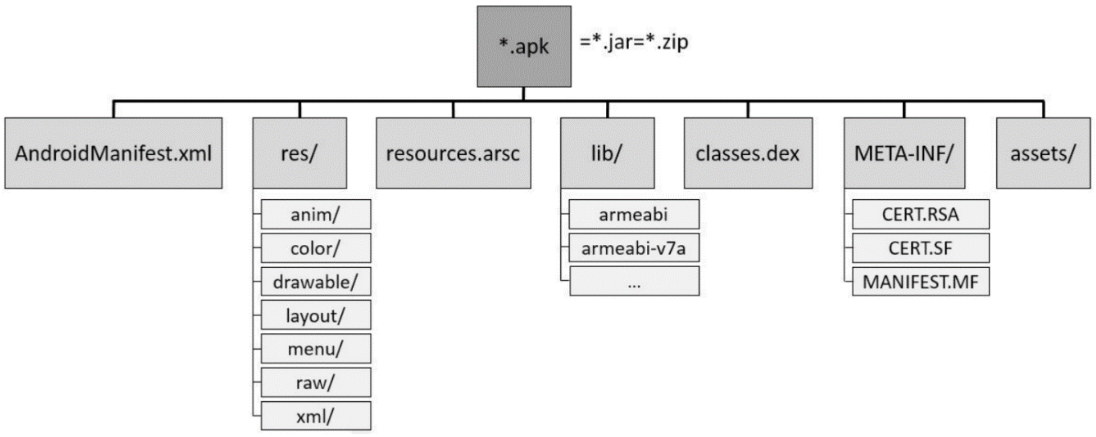


Figure 19 - APK file structure [100] (draft version – not published)

Dalvik Executables (DEX files)

DEX files (cf. classes.dex above) contain [64] the actual program logic and are structured as presented in Figure 20 with typical information such as an identifier (Magic), a checksum, its file size, offset information to Strings and lots of other properties [126, p. 12ff]. Detailed knowledge on the format is of interest to advanced reengineers only, and perhaps those who encounter junk bytes on older Android versions. Instead, most reengineers will benefit from existing tools that convert DEX files to high-level assembly (see 8.2.2). Therefore, further information on the fields and structures of Figure 20 may be found in [126] [64, p. 14ff] and are of no imminent interest to the reader and also not in terms of this dissertation.

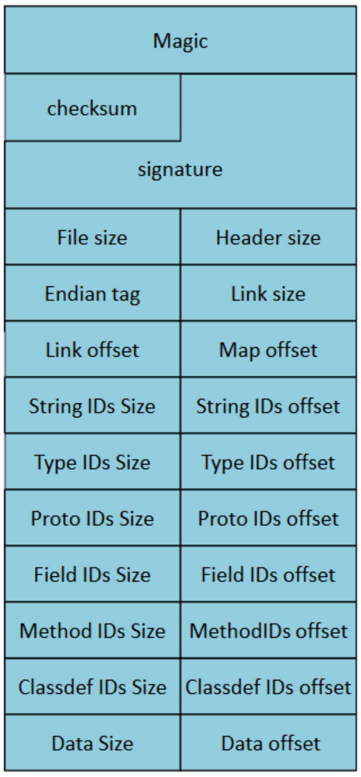


Figure 20 - DEX file format [126, p. 12]

ODEX file

ODEX files are “Optimized Dalvik Executables” [75, p. 23] only, and generated by Android for improved runtime execution.

ART file

The ART file is used on newer Android versions and is “an image file with a heap of pre-initialized classes and objects”[33] [122]. Its code may be called by the following OAT files [127]. Actually, there is only one ART file called boot.art [117, p. 11].

Optimized Ahead of Time file (OAT file)

OAT files [75, p. 25ff] are used on newer Android versions and the aforementioned DEX files are converted using the tool “dex2oat” by Android. It’s basically “An ELF [‘dynamic object’ [127]] file with DEX code, compiled native code and metadata” [34] [122]. OAT files are structured as shown in Figure 21.

[33] transcript by author
[34] transcript by author

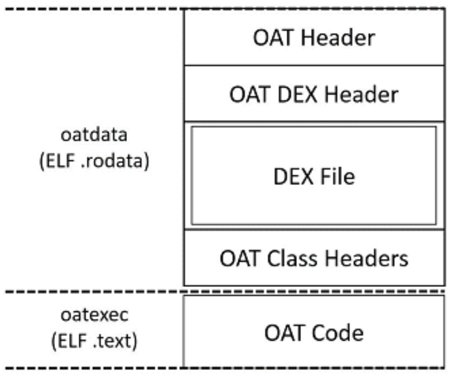


Figure 21 - File format of an OAT file [100] (based on [123])

An interesting fact is the stored DEX file(s) (see Figure 21) within the OAT file, in addition to the actual OAT code. While that native code might be the better approach in terms of securing code against reengineering, we found out that current Android versions still require the DEX files (e.g., for debugging purposes [122]).

There are several OAT files used on Android. The major one is boot.oat and contains the frameworks. In addition, each application provides its own OAT file stored in the former ODEX file [117, p. 11]. Further information on the OAT Header and OAT DEX File Header may be found in [127] and [117].

7.3.10 Compilation of Android apps

The summarized procedure, as shown in [128] and outlined in Figure 22 in detail for the compilation of Android Apps, is the compiling from Java source codes to class-files that get cross-compiled to the Dalvik Bytecode. It is named the Dalvik Executable or DEX file. In addition, all resources, references as well as the Android Manifest file are collected and stored together with the executable in an application package file (APK file) that gets signed by the developer key for further distribution on app markets. The shown R-file (in Figure 22) includes references to resources, while aidl is Android’s Interface Definition Language [129] to allow the usage of RPC services (optional). Moreover, native libraries get included almost at the end (see “Other Resources” in Figure 22), before the whole package gets finalized and signed. The signature is a security feature as well. It should prevent malicious replacements of apps on a phone, while it can also be used to detect modifications.

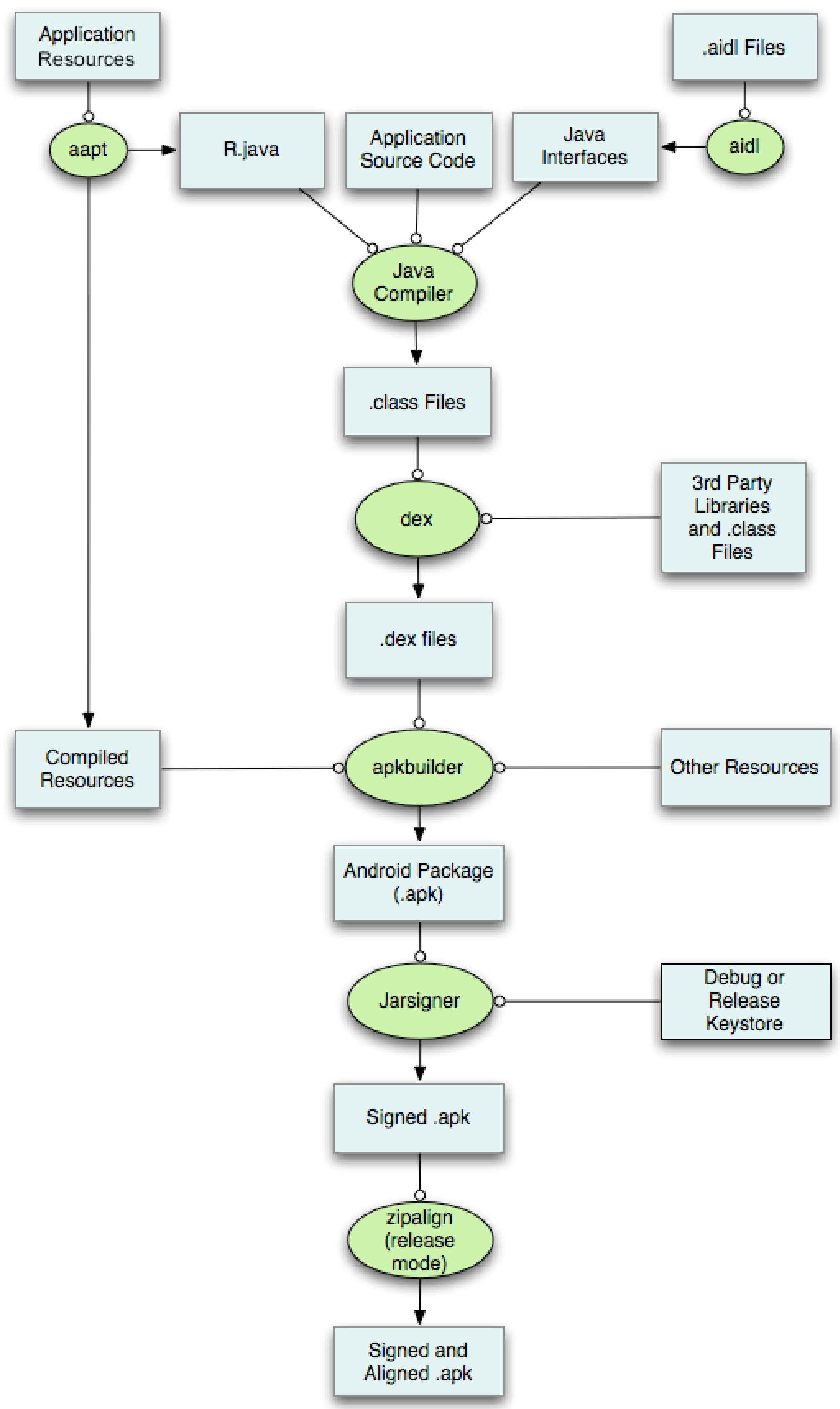


Figure 22 - Building APK files [128]

7.3.11 Compilation of Android apps using native code

A special case is the usage of native code (C/C++) in Android Apps that requires special tools (Android NDK) and IDE configurations as outlined previously. In this section, the compilation process, as shown in Figure 23 is explained. Developers are required to create a Java Native Class within their Android Project in the jni-folder that can be compiled to a class-file and used to create the C-header file, which is also the base for the actual C/C++ source file. In addition, two make-files are required, which may define further configuration settings, e.g., to be included libraries (e.g., LOCAL_LDLIBS := -llog for integrating Android's logging feature). Then the NDK tools[35] can be used to compile and link the code into a shared library (*.so). The compilation process is almost independent from Android's app compilation and the SDK only includes the most recent shared libraries into the package file (APK) upon compilation of the Android app itself (see "Other Resources" as shown in Figure 22, see 7.3.10) [130].

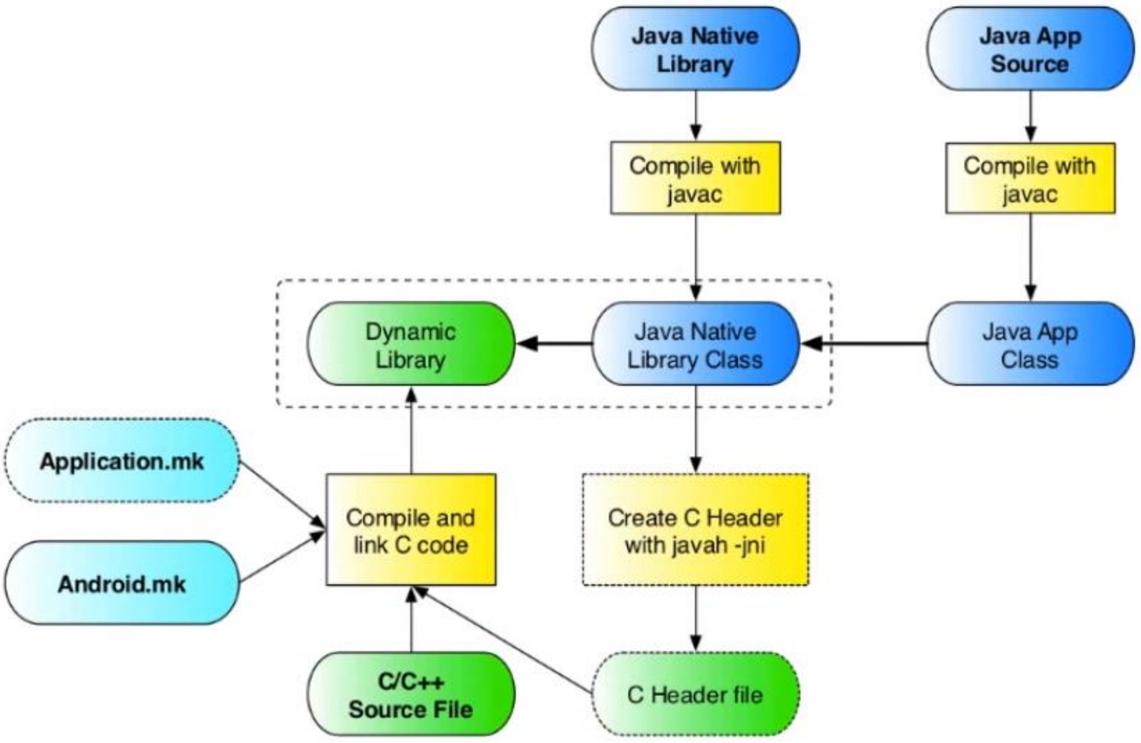


Figure 23 - Build process of native code using the NDK [130]

[35] Command: ndk-build all

7.3.12 Installation of Android apps

As described in [100, p. 8ff], an APK file is handled by Android's Package Manager that copies the original file to a file named base.apk stored in the directory /data/app/ and there within a directory called after its app name, in addition to an appended "-1" (see Figure 24 below). Furthermore, the native libraries are copied to that directory, too. Depending on the used Android Runtime, the dex2oat (ART) or dexopt (Dalvik VM) tool is used on the extracted classes.dex file and its output is stored in /data/dalvik-cache/<arch>/ using the format "data@app@<packagename>.<appname>-1@base.apk@classes.dex" [100]. It is important to mention that this file is actually an ODEX file (DVM) or OAT file (ART) internally. Finally, the Package Manager adds an entry to the files packages.xml and packages.list within /data/system that contains meta information like UID/GID or required permissions.

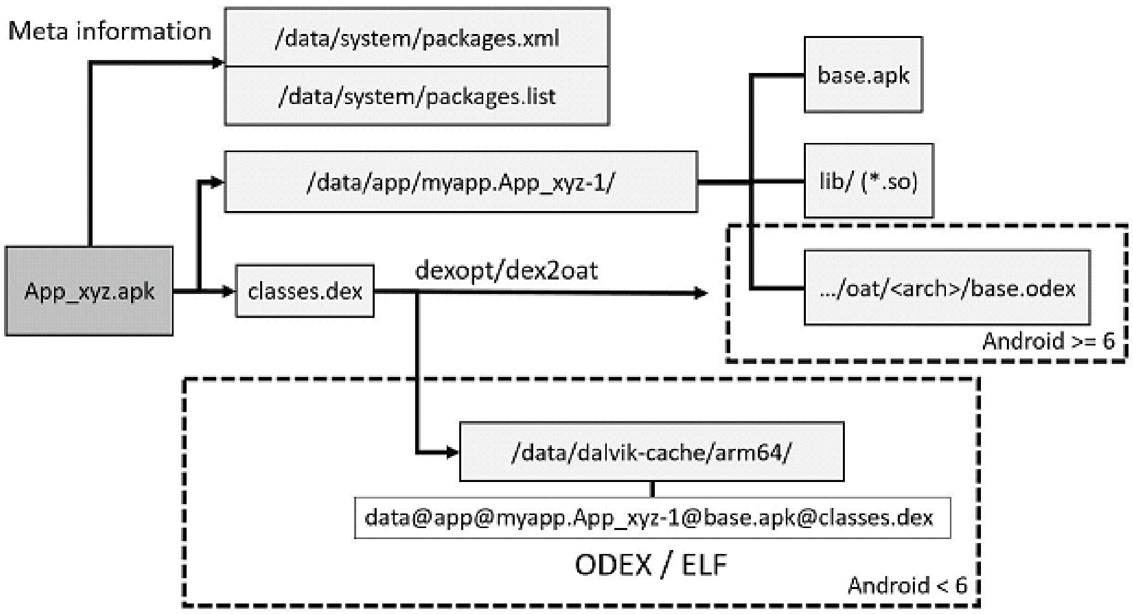

Meta information
/data/system/packages.xml
/data/system/packages.list
/data/app/myapp.App_xyz-1/
base.apk
lib/ (*.so)
App_xyz.apk
classes.dex
dexopt/dex2oat
.../oat/<arch>/base.odex
Android >= 6
/data/dalvik-cache/arm64/
data@app@myapp.App_xyz-1@base.apk@classes.dex
ODEX / ELF
Android < 6

Figure 24 - Installation procedure [100, p. 9]

7.3.13 Execution of Android apps

The basic execution of Android Apps takes place as follows [100, p. 10] (based on [120]):

The entry point for the execution is the ODEX file that either contains optimized code for the Dalvik VM or pre-compiled code for the ART VM. Upon request, and by a click on the icon by the user, a forked version of the Zygote process launches so that the created child process (the app) inherits several resources and loaded libraries of its Zygote parent process (Notice: This is also how Xposed is injected into apps [131]), while app-specific resources or libraries are additionally loaded.

Furthermore, native libraries of the app are linked either with the created executable of Dalvik's JIT compiler or the native code version of that app in case of the ART VM. Figure 25 illustrates that process in a simplified diagram.

Additional details on the startup process and Zygote's deep internals may be obtained from [100, p. 22ff].

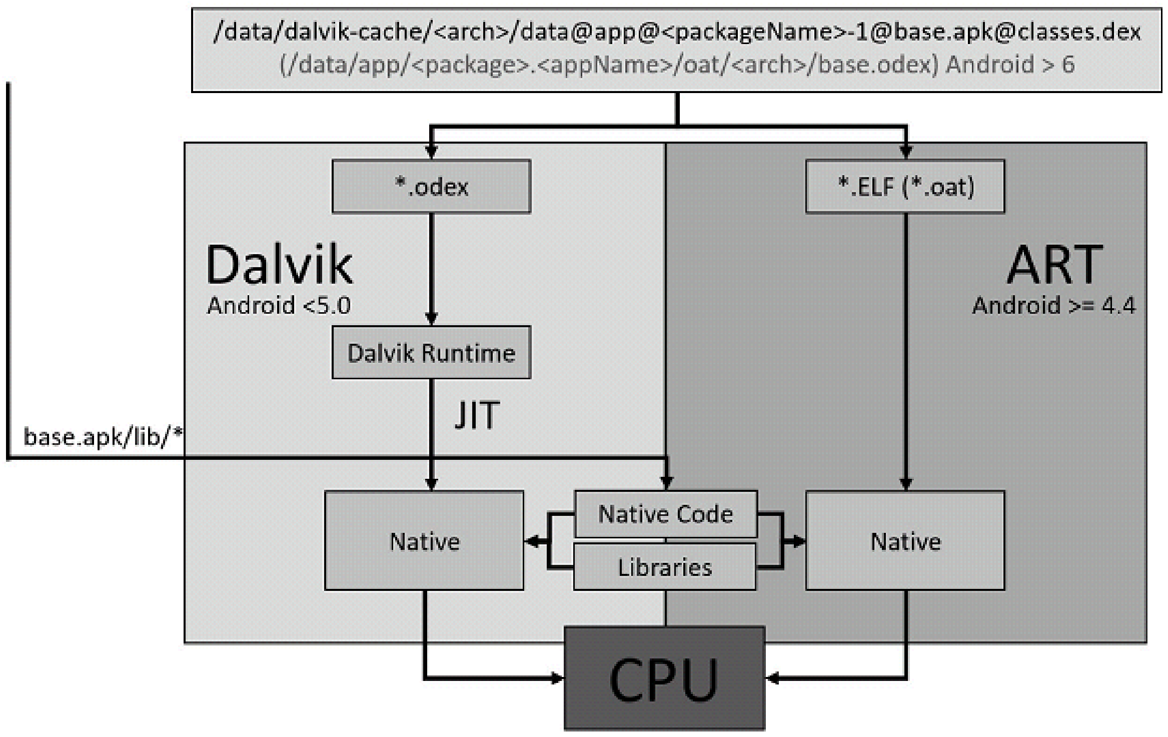

/data/dalvik-cache/<arch>/data@app@<packageName>-1@base.apk@classes.dex
(/data/app/<package>.<appName>/oat/<arch>/base.odex) Android > 6
*.odex
.ELF (.oat)
Dalvik
Android <5.0
ART
Android >= 4.4
Dalvik Runtime
JIT
base.apk/lib/*
Native
Native Code
Libraries
Native
CPU

Figure 25 - App Execution by Dalvik- and ART VM [100, p. 11]

7.3.14 Lifecycle of Android apps

Android provides several components and one of them is the activity. Basically an activity represents the screen content a user is watching on the display. It may consist of layouts, buttons, pictures and other so-called view objects that can be added to an activity and ultimately shown to a user [132]. In terms of the proposed copy protections (cf. 11.4.4) some knowledge about the lifecycle of activities (see Figure 26) is important. For instance, thinking about the best functions on where to initialize variables, while observing the specialties of Android that a background activity might get destroyed upon memory requirements by other apps.

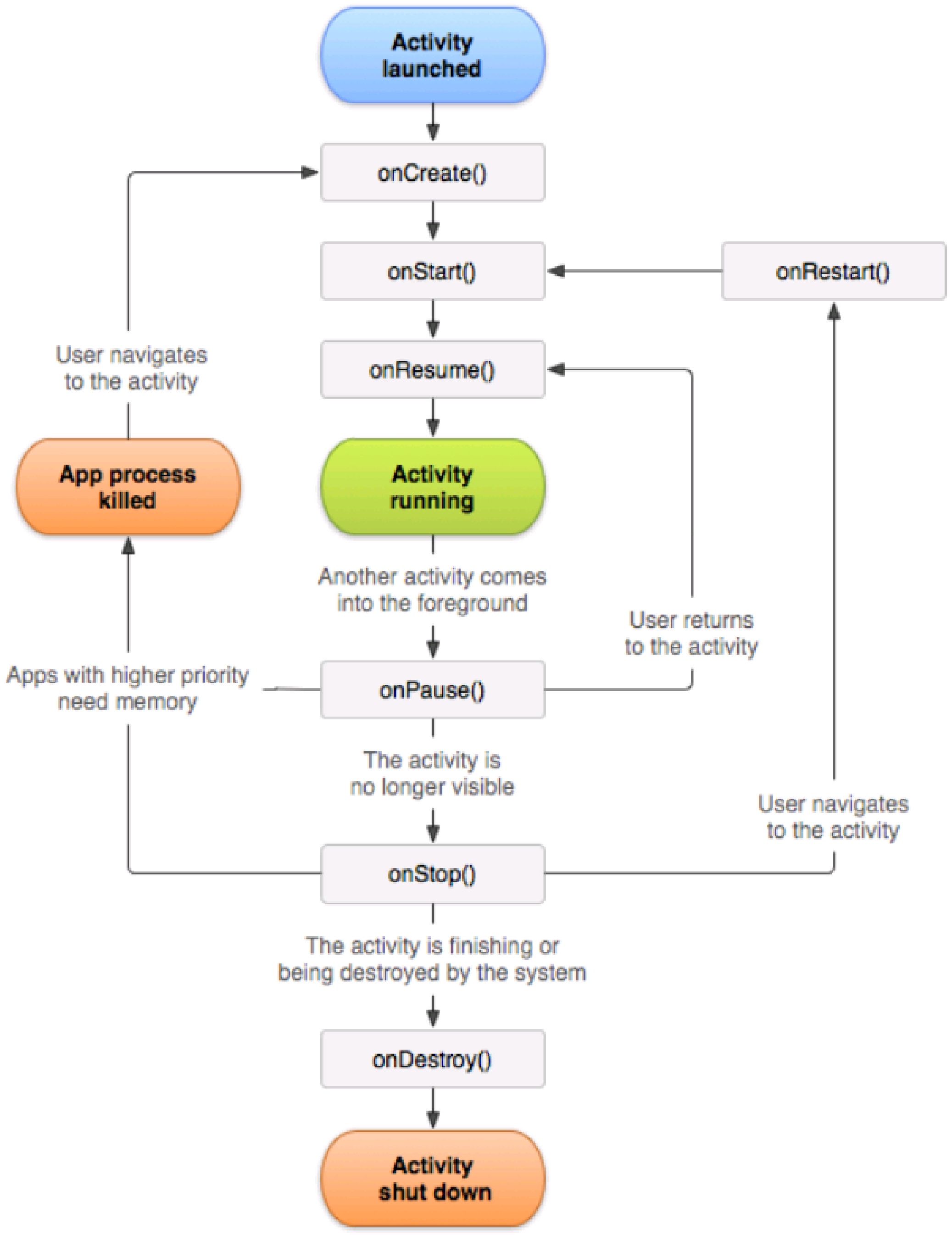

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 26 - Lifecycle of an Android Activity [132]

8 Topic-specific background

Due to the focus of this dissertation on copy protection and related issues the current chapter is dedicated to Android reengineering and its fundamentals. First, related terms are introduced, before the used assembly dialect for Android called “smali” [133] is presented in more detail. In addition, the currently available tools for reengineering purposes are shown, and those ones to prevent it. Also, the basics and history of copyright protections for mobile and desktops operating systems are introduced as well as some fundamental information on attacking options. Moreover, existing solutions in software and hardware for data protection are introduced.

8.1 OP codes, mnemonics and related terms

As described in [134], an **OP-Code** or operation code is a number representing a machine command that gets executed by a machine (processor) or virtual device, e.g., 0x32 represents the command if-equals on the Dalvik VM [116].

Instead, a **Bytecode** often refers to a virtual machine or interpreter only [134].

The human-readable representation of such an OP code, e.g., if-eq for 0x32 [116], is called **mnemonic** [134].

A group of these OP codes with their parameters is called **Assembly** again [134].

Finally assembly source codes get compiled to a **machine code** that is basically an ongoing formation of numbers only [134].

8.2 An introduction to smali (assembly)

“Smali” is the “Icelandic equivalent […] of ‘assembler’” [133] and represents a programming language that results from reengineered apps by using the tools smali and baksmali. Developers familiar with any assembler dialect as well as Java in general will be able to adapt to that language within a few days, while editing the source code might require additional studying. The following sub-sections (based on [133]) will explain the most important commands and structures in a quick summary. Further details on the actual OPcodes may be viewed in the Dalvik documentation (cf. [135]).

Registers [136]

“Registers are always 32 bits [in size], and can hold any type of value [while] 2[sic!] registers are used to hold 64 bit types (Long and Double)” [136].

Registers may be separated into local and parameter registers:

- v0 and v1 are the first and second local registers
- v2/p0, v3/p1 and v4/p2 are the first, second, and third parameter register

The amount of registers used by a function may be specified in two ways:

- “The .registers directive specifies the **total** number of registers in the method” [136]
- “The […] .locals directive specifies the number of **non-parameter** registers in the method**”** [136]

Variables [137] [136]

Variables are usually represented by a register that holds the corresponding value, e.g.,:

const/high16 v0, 0x1

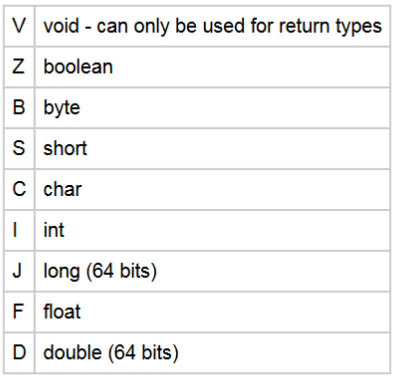
const/high16 v1, 0x12

“const-wide/high16 v0, 0x4014“ [138]

Also, it is important to know that float and double values are stored in two consecutive registers, while the smali code seems to address one register (see example above), only [138].

Data Types and Primitives [139]

The primitives and data types are defined by Oracle as follows [140] and are also used by smali.



V	void - can only be used for return types
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)

Figure 27 – Primitives [139]

Objects are indicated by an L, e.g. Ljava/lang/String represents a string object.

Arrays, such as an integer array, are represented by leading brackets that define the dimensions, e.g., [I means a single dimension integer array. Of course, the same applies to objects.

Functions [139]

Functions are defined in a detailed way by its name, types, parameters, and return values, e.g.,

Smali: Ljava/lang/String;->getBytes(Ljava/lang/String;)[B

Java: ByteArray = SomeString.getBytes(“UTF8”);

8.2.1 Dalvik bytecode and its general issues

As already explained earlier, Dalvik Bytecode is a compressed and restructured version of usual Java Bytecode. Initially, the Java source code is compiled to Java Bytecode and afterwards converted to Dalvik Bytecode [141].

Due to the included references, the resulting assembly code (smali code) can be much more easily understood in comparison to disassembled native code (ARM binaries; see 10.3ff for a native code evaluation). Table 2 shows a code snippet of an example for a decompiled Android application using the APKtool[36]. It is probably the most commonly known reengineering tool for Android. It requires little practice and some knowledge about the used data types[37] by Java/Oracle only to allow skilled developers to understand a code's meaning (e.g., I equals Integer and V means void as already previously outlined). The reengineered assembly code includes even the names of variables and functions by default. It represents the Java source as shown in Table 1.

```
 […]
setContentView(R.layout.main);
int a, b;
a = b = 5;
[…]
```

Table 1 - Java Source Code Snippet (based on [106])

```
[…]
# a function gets called with an integer and has no return value (void)
invoke-virtual{p0, v2},Lde/tum/EasyApp/EasyApp;->setContentView(I)V
.line18
const/4 v1, 0x5        #   value of 5 stored in register 1
.local v1, b:I         #   the name is b and of type integer
move v0, v1            #   it is copied to register 0
.line 19
.local v0, a:I         #  the name is a and of type integer
 […]
```

Table 2 – Smali Source Code Sample (based on [106])

As anyone might notice, it is easily possible to change values or even the code (logic) itself and recompile everything. Therefore, we can summarize that it is (with practice) fairly easy to reengineer and modify Android applications as previously stated in a related paper by the author [142].

[36] Reengineering Tool from https://code.google.com/p/android-apktool/

[37] https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields

8.2.2 Available tools used for reengineering / modifications / hacking

This section represents a selection of tools important for reengineering. One of the first tools available to researchers and hackers to reengineer Android Applications was the previously introduced Smali and Baksmali with its equally-named tools combined in a tool collection [133]. Other tools are based on this tool collection (e.g. APKtool), while other programs focus on a specific issue to support the reengineering by other tools again (e.g. dex2jar or dextra). Figure 28 shows an overview on the most common tools, while these tools and further ones are explained below.

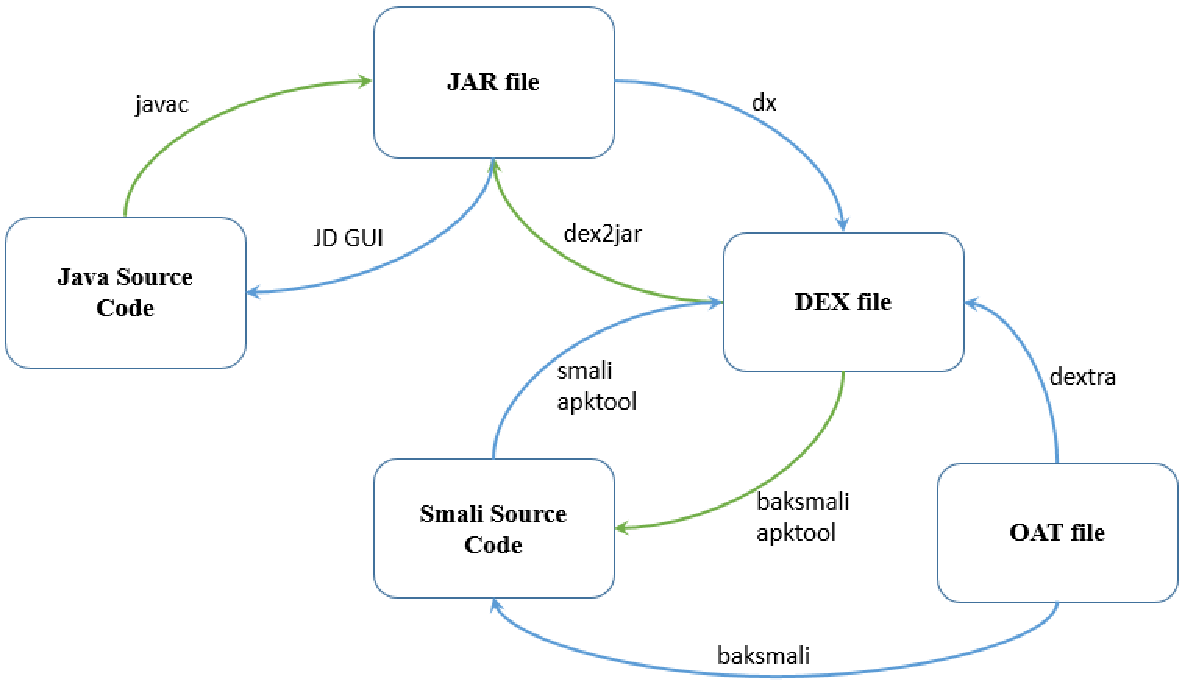


Figure 28 - Overview of De-/Compilation and convertation options by example (based on tool information sources below)

Smali / Baksmali

Smali/Baksmali [143] is an Assembler-Disassembler tool collection to reengineer Android apps. It requires the classes.dex file included in every APK-file (or OAT-files nowadays [144]). The resulting code is a type of a high-level assembler code that still includes lots of references by default. The "syntax is loosely based on Jasmin's/dedexer's syntax" [143]. An example for smali code may be found on the previous page.

APKtool

The APKtool [145] is used to disassemble and assemble Android apps with all additional resources. It is based on the aforementioned tool collection. It provides a debugging feature and is able to reengineer the APK-files and to rebuild them after any modifications.

DEX2JAR

DEX2JAR [146] transforms DEX-files to JAR-files. It is meant to convert DEX files used on Android to the different-structured JAR format to allow further processing of that file with reengineering tools available for usual Java, e.g., a Java Decompiler.

JD-GUI

JD-GUI [147] is an easy to use Java-Decompiler. JAR files may be dragged on the application for instant decompilation. The resulting code most often reveals the logic, but is not well composed to be used for future compilation again.

AndroGuard

Similar to the APKtool, AndroGuard [148] is a tool to decompile Android apps. In addition, it offers some additional features like call graphs that enable reengineers to understand code more easily.

Virtuous Ten Studio (VTS)

VTS [149] is an IDE specially designed for reengineering of Android smali-code. It provides highlighting and instant help.

Online Decompiler

There are also cloud-based solutions available that combine the aforementioned tools in a web service to provide users an even easier reengineering solution. For example, it provides all smali- and java-sources in one easy step like http://www.decompileandroid.com.

Xposed Framework

The Xposed Framework [150] is used on rooted Android devices to modify any Android applications on-the-fly. It is possible to intercept methods and to change any values. It is often used to add additional functionalities or to remove size limits. For instance, in our research we used it to circumvent Android’s License Verification Library by exchanging the necessary parameters in its functions to simulate a valid license (see 10.1.5).

Cydia Substrate

The tool Cydia Substrate [151] was released for Android a few years ago. It provides similar functionality like the Xposed framework to intercept and manipulate Java code, while also allowing the same for Android native code (cf. Android NDK). Unfortunately, so far it is not supported by newer Android versions and it was updated in 2013 [152] for the last time.

Frida

Frida [153] is a framework to intercept processes of various operating systems including Android. It uses JavaScript and Python.

Lucky Patcher

Lucky Patcher [64] is not a typical reengineering tool, but more of a generic cracking tool to circumvent (illegally) any copyright protections and to remove ads in Android apps. For cracking apps, it provides several modes. An analysis and more detailed description is found in 10.1.3.

Dextra

Dextra [154] is developed as a better alternative to the standard dexdump tool by Google. In addition, it has the unique feature to extract DEX files from OAT files, which is an important fact for reengineering on modern Android versions; however the current ways to gather DEX files from APK files are still available and require an usual unpacker only (cf. APK file = ZIP file).

Tool suites

In addition to the single tools outlined above, there are collections of tools available and included in special purpose Linux distributions. One of them is Santoku Linux[38] that focuses on Mobile Security and all related topics including reengineering and forensics. Another tool is Bytecode Viewer[39] that integrates several decompilers and others tools. It may be called an all-in-one tool for reengineering.

Tools for native code

In general there are lots of reengineering tools for analyzing binaries (cf. libraries created by the Android NDK) available [155] like IDA[40], Hopper[41], ODA[42] or the online Retargetable Decompiler[43]. Native code is much harder to reengineer and most often, there are no longer references available. Chapter 10.3ff offers a more detailed analysis.

Further information and tools

While the list of tools available is still not completed, further information (as well as more detailed information) are shown in the slides provided by Tim Strazzere and Jon Sawyer in [156].

One of the best compendiums, with a focus on security and that introduces hundreds of tools, books, and talks is the Mobile Security Wiki (see [157]) by Philippe Arteau et al.

8.2.3 Available tools and options to extend time on reengineering

Preventing reengineering is simply not possible and as long as customers have access to the hardware or software it is only a matter of time until someone cracks a certain protection. Nevertheless, it is possible to extend this time. A common method for Java and therefore for Android as well is the usage of obfuscation. Typical and general obfuscation methods as outlined in [158] include features to prevent the debugging like methods for identifying emulators or virtual machines, methods to detect or even prevent debuggers, options to prevent actual disassembly as well as protection against tampering. Further possibilities are virtualization-obfuscation that uses emulation to execute a random instruction set to prevent disassembly [159]. The usage of packing tools (polymorphism) is also a common way to protect actual code besides adding a lot of nonsense to confuse attackers (metamorphism) [158].

Focusing on Android and since decompiled Java code still contains many references (cf. 8.2.1/ smali code) by default, an obvious method to increase the difficulty on reengineering is to remove all possible names and references. Further methods might be the encryption of resources

[38] https://santoku-linux.com/about-santoku/
[39] https://github.com/Konloch/bytecode-viewer
[40] https://www.hex-rays.com/products/ida/
[41] http://www.hopperapp.com/
[42] https://www.onlinedisassembler.com/odaweb/
[43] https://retdec.com/

and all used strings. As a side effect, the optimization also decreases the file size. Tools that need to be mentioned providing these mechanisms are ProGuard and its commercial version DexGuard [160] [161].

In addition, researchers invented further methods for obfuscation on Android. An interesting approach (see details in 9.2.2 about "Divilar") is the obfuscation by exchanging all opcodes, while restoring them on-the-fly upon execution. This approach prevents the protected app from being decompiled by all typical reengineering tools [162], while unfortunately this approach can no longer be used effectively on modern Android (cf. pre-compilation requirement).

Taking the easy reengineering options of Dalvik Bytecode (resulting from Java code) in mind, a suggested option could be to use native code (e.g., Android NDK) for security related tasks, since it is commonly known that it is much harder to reengineer. This assumption is even confirmed by Mr. Kralevich[44] stating "Speaking for myself (not Google). […] I agree that native code is more resistant to reverse engineering, so it's likely more secure for your copyright protection mechanism" [163]. In general Google recommends using Java code instead, since handling native code is more difficult [7].

In addition, there are several other options as mentioned in the related work section including, e.g., encryption and dynamic code loading solutions.

8.3 Basics on copyright protection

The following section covers the basics in terms of copy protection in general, in addition to its history for both the desktop and mobile world and available methods by example.

8.3.1 Copy protections on desktop computers

As previously mentioned in the introduction section, and since the early days of computing, content- and copyright protection have existed. Also, the used methods improved over time to compete with available technologies in order to provide a reasonable protection against software piracy.

In general, one can assume the following reasons for using copyright protections [75] [50] [164]:

- Loss of money due to piracy
- Repackaged applications might be modified to gain interests for the attackers
- Customers are at risk, since repackaged applications might contain Trojans etc.

At the beginning in the early 1990s, so called "code wheels" [165] were used. Most of the time they consisted of two paper-disks that were moveable in order to calculate different results. Gamers[45] were required to solve a type of riddle using that tool. At that time, it was still difficult

[44] "Nick Kralevich is head of Android platform security at Google and one of the original members of the Android security team" [358]
[45] refers to computer users, who are gaming

and expensive to produce copies of these disks, and scanners and copy machines were not as readily available like they are today, not speaking about missing internet possibilities by that time.

Nevertheless, the companies noticed quickly that these ideas were not sufficient to protect against illegal copies and they came up with more advanced ideas that required special hardware or software. Again, this was to fulfill a reasonable protection for that time. Examples for these types of protection as outlined in [36] are as follows:

- The usage of additional floppy disk tracks out of scope (not copied by default copy commands) to place additional data not found on copied disks anymore (cf. “40 tracks on a 5.25" disk, and 80 tracks on a 3.5" disk” [36])
- The usage of intentional disk errors or other damages not copied with the default OS copy commands, and therefore, copies (without these errors) are easily identified
- Custom formats to hide the data and prevent DOS from accessing any files
- By using “weak […] fuzzy […] [or] strong Bits” [36] that influence physical properties and may be detected by a disk drive / the copyright protection methods

There are several other possibilities found in [36].

Then, by around 1995 [166] [167], Microsoft released its famous operating system “Windows 95” [166] that started a new era for home computers, since the initial version of “DirectX”[46] [167] was released. Moreover, slowly the gaming industry adapted to this new operating system. At the beginning of this new era many games were still developed for DOS as well as Windows (e.g., Westwood’s Command and Conquer: “Red Alert”/two binaries for each OS on the same CD) and disk space was still limited to a few gigabytes [168]. While it appeared impractical to copy these huge CDs — each one around 700MB in size — by that time, most games on CDs already had copyright protection. Examples of protection methods are SafeDisc, SecuROM and LaserLock [169]. Similar to the known and presented methods for the floppy discs above, these techniques made use of manipulating special properties of discs once again. For instance, LaserLock [40] does this by not only encrypting the original executable and adding its security code, but it also manipulates the disc format by adding data in hidden areas to apply a physical signature to the disc, which is verified later by the protection method [40].

While the quality of the games (especially graphics) improved over time and their file sizes quadrupled with the introduction of at least DVDs and BluRay Discs, the copyright protections adapted themselves to the new formats only and, e.g., LaserLock was also applied to protect DVD-based software [170].

While games are still available on DVDs — commonly known as “boxed […] games” [171] — an increasing number of software is also sold on online platforms nowadays [171]. At the beginning, producers sold their own games on these platforms only (e.g., Steam by Valve), but one can find almost any game at these online shops today (e.g., Steam and Gamesload.de). Of course, these downloadable games do not require any physical discs anymore and therefore lack the typical copyright protection mechanisms. Instead most of these games are shipped with a registration or activation key that is linked to a user account [172]. For online multiplayer games it is most often an essential requirement now.

[46] DirectX provides developers multimedia APIs and is most often used for games

This new approach is quite convenient for users as well as companies. While users may benefit from a ranking system among games (cf. Battle.Net for games like Starcraft[47] and Steam for games like Counter-Strike[48]), they can frequently download games according to the license, and still, even after years, by using their credentials only. A slight disadvantage might be the fact that these games usually require a permanent internet connection. In contrast, the companies often acquire usage information and customers, who use games without their credentials (besides linked activation keys) are frequently unable to use the games anymore, because using the server infrastructure is limited to legal accounts. This is a great advantage for copyright holders. Exceptions are single player games that do not require a permanent internet connection for gaming. These types of games still face the danger of being cracked and copied illegally, as addressed in 8.4ff.

In addition, pricy software products, or those that require special protection (e.g., the IDE for G&D's MSC) are using so called dongles to protect their applications [41]. In its most basic version, a hardware dongle may be represented by a small device that is attached by a USB or another interface with special wiring inside. Nevertheless, modern dongles may include a small microcontroller to deliver encrypted data or keys to an application to fulfill some sort of handshake [173].

8.3.2 Copy protections on mobiles

When observing the mobile world with its app markets, one can find a similar approach to the recent example in the desktop world. This includes the usage of an account (e.g. Google account, Amazon account) that is linked to all installed and bought apps and is combined with a simple protection mechanism to check that an executed app belongs to a legitimate account before other parts get executed. In fact, it is not a real copy protection, but a license verification only. The app itself can be copied, but does not work on other devices anymore. Depending on the app, market-providers like Google or Amazon offer different solutions here and are presented next in more detail.

Google's "License Verification Library"

The "License Verification Library (LVL)" [47] released in 2010 [174] provides basic protection for developers. Only apps supplied through the Google Play Store[49] are covered and that requires the Google Play services to be installed, while preventing the app from running otherwise [66, p. 20]. This allows developers an easy solution to integrate basic copyright protection (actually license verification) into their apps that sell through the Google Play Store. Internally, an application uses a method call while implementing a callback to handle the actual license response by Google. The communication with the Google servers is arranged by the Google Play client that gets involved by a remote IPC[50] request from the app. The usage of a nonce, as well as public/private key procedures (using RSA[51]), should ensure a safe and valid response. Only Google knows the private key to sign any responses, while the app can verify it

[47] PC game - http://us.blizzard.com/en-us/games/sc/
[48] PC game - http://store.steampowered.com/css
[49] Google's Platform for offering apps - play.google.com
[50] Inter-Process-Communication
[51] Encryption by Rivest, Shamir und Adleman (Inventors)

by using the embedded public key in the app (included by the developer). The Google Play client provides the Google servers basic user information for identification purposes and validation of the license request, too. Figure 29 illustrates the basic implementation [64, p. 55ff].

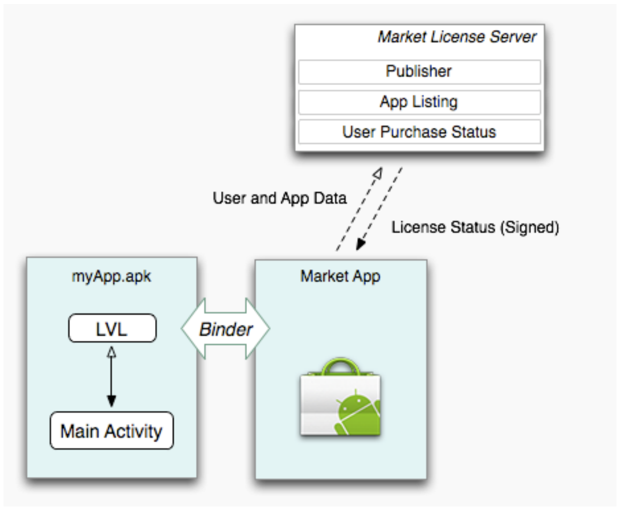


Figure 29 – How Google's License Verification Library works [47, p. top]

A sample and minimal implementation is provided below as described in [66, p. 18ff]. Besides the basic requirements such as a Google Publisher Account for developers, the app needs to sell through the Google Play Store, and users' devices need to have the Google Play Services installed. For integrating the LVL in an app, it needs to acquire permission to use the licensing service first. In addition, the app's public key needs to be fetched from the Developer Console. It must be integrated into the code snippet of Figure 31. Here is also where the basic configuration takes place, by providing the LVL a unique Android ID that is a "64-bit number (as a hex string) […] randomly generated when the user first sets up the device" [175]. A salt consisting of random bytes, is also required in addition to the package name. The 'AESobfuscator' here is used to store license responses hidden. Finally, the actual license request is triggered by passing the callback. Of course, in advance the callback methods - as shown in Figure 32 - need to be integrated into the own code providing the implementations to the cases/functions applicationError(), dontAllow() and allow().

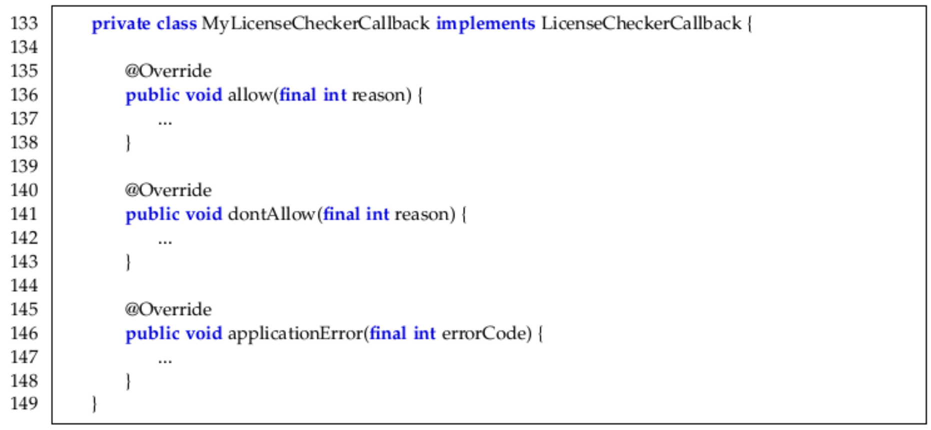
```
...
<uses−permission android:name="com.android.vending.CHECK_LICENSE" />
...
```

Figure 30 - Permission to use LVL [66, p. 18]

```
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);
```

Figure 31 - LVL Configuration [66, p. 19]

The details of these methods are found in Figure 33. "The applicationError() [method] 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" [66, p. 18] (based on [176] [47]). Additional details may be found in [47] and [177] as well as in our analysis chapter (see 10.1.5). Even further internals are outlined in 11.4.8 obtained by reengineering of Google's services.



```
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) {
        ...
    }
}
```

Figure 32 - LVL Callback methods [66, p. 19]

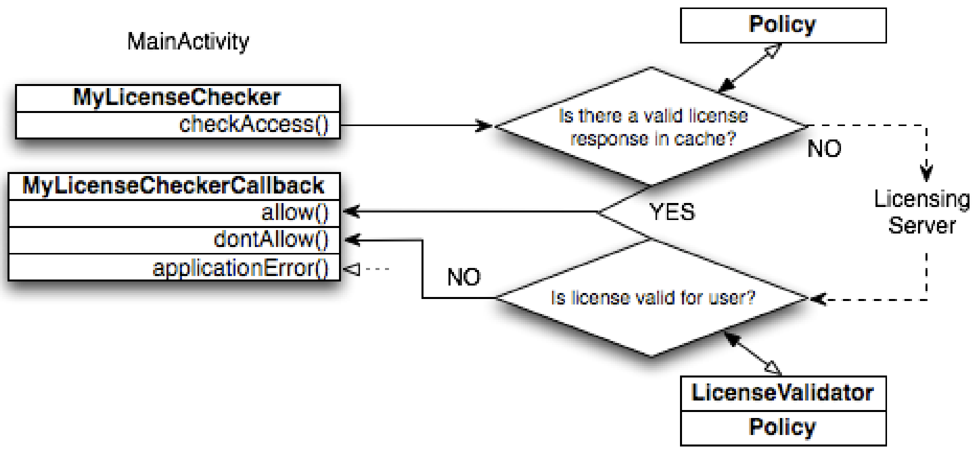


Figure 33 - Overview license check [177]

Amazon's DRM

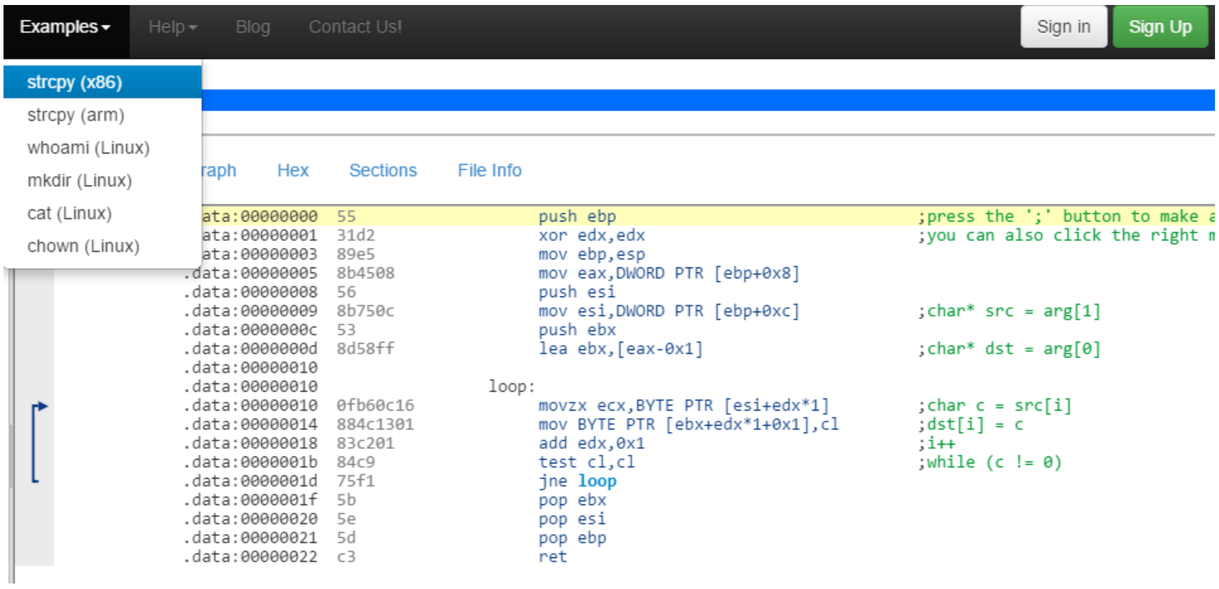
Amazon's approach [64, p. 23] within its Amazon App Store, is different from the one presented by Google. While developers of Google's LVL are required to integrate and modify it on their own, Amazon applies its own protection mechanisms upon the upload of an APK file by decompiling and repackaging the app, and while adding and modifying the code as well as applying a new signature that is unique for the developer [46]. Details on this protection are found in 10.1.4.

SlideMe's SlideLock

SlideMe [65] is an alternative, but rather small, app market similar to Amazon's AppStore and offers its own license verification and service. Therefore, developers are required to integrate a jar-based library into their app. It is based on identifying a device by using either IMEI or the WiFi MAC address, besides requesting license information from a license server. It features, e.g., periodical checks and leaves it up to the developer to define actions such as a grace period, e.g., upon travelling of the user with no available internet connection.

8.4 Basics on attacks on copyright protection

For understanding and ultimately cracking copyright protection the reengineering of an application is an essential first step. Nevertheless, one huge difference is the format of applications on different architectures. While most applications and protection-drivers in the desktop-world are likely only available as native code (e.g. x86 binary code), its reengineering may be considered extremely difficult due to missing references and pure assembly code, while applications on Android are available as Dalvik bytecode instead. Figure 34 shows an example for a reengineered x86 application of the strcpy function. It is compared to the Dalvik Bytecode example in Table 1 and Table 2 in 8.2.1. Here, one can clearly recognize the differences of defined variables and included references in the smali-code in comparison to the x86-assembly-code (blue) below (Notice: the explanations in green usually are not available and provided by the author of the website).



```
ata:00000000  55              push ebp                           ;press the ';' button to make a
ata:00000001  31d2            xor edx,edx                        ;you can also click the right m
ata:00000003  89e5            mov ebp,esp
.data:00000005  8b4508          mov eax,DWORD PTR [ebp+0x8]
.data:00000008  56              push esi
.data:00000009  8b750c          mov esi,DWORD PTR [ebp+0xc]        ;char* src = arg[1]
.data:0000000c  53              push ebx
.data:0000000d  8d58ff          lea ebx,[eax-0x1]                  ;char* dst = arg[0]
.data:00000010
.data:00000010              loop:
.data:00000010  0fb60c16        movzx ecx,BYTE PTR [esi+edx*1]     ;char c = src[i]
.data:00000014  884c1301        mov BYTE PTR [ebx+edx*1+0x1],cl    ;dst[i] = c
.data:00000018  83c201          add edx,0x1                        ;i++
.data:0000001b  84c9            test cl,cl                         ;while (c != 0)
.data:0000001d  75f1            jne loop
.data:0000001f  5b              pop ebx
.data:00000020  5e              pop esi
.data:00000021  5d              pop ebp
.data:00000022  c3              ret
```

Figure 34 – Example (blue code) for a disassembled x86 code using the online disassembler for strcpy [178]

This issue even applies to the latest Android versions using the ART Runtime, since they still embed the DEX file (as indicated in 2016 by [100] (based on [123] [124])) in the OAT file (see Figure 18 in 7.3.8). As already presented in 8.2.1 the reengineering of the Dalvik bytecode (DEX file) is much easier and may even include references and names depending on the used obfuscation method (if any was used at all).

In general, all protection mechanisms face the risk of getting cracked after a while. Here we need to distinguish between the desktop-world and the mobile-world again, as well as different protection mechanisms and its countermeasures.

8.4.1 Cracking methods on desktop computers

Cracking floppy disc protections

Back in the old days and recognized by the author himself, copy-protected software was most often cracked by special cracking tools that replaced the desired patterns within executable and specific versions of a program. Either the software pirates shipped a special tool (called crack) or they provided the cracked binary already. Since the internet was almost unknown at that time, it was much harder for users to obtain any illegal copies or cracks. Pirated software was mainly shared among groups and friends, or friends of friends, via physical floppy discs.

Cracking CD/DVDs/etc. protections

With the rise of modern multimedia computers, Windows95/98, CDs, and the Internet, the situation slowly changed around 1995 to the end of the millennium. Manufacturers started to deliver their software on CDs and updates over the Internet happened more frequently. Also, software pirates probably got annoyed by the constantly updated executables that required new cracks for each version. In addition, cracked software often lacked features such as the background music on the CDs, and cracks most often only disabled the checks for a legitimate CD, while the so called key-generators delivered any required license codes. In fact, people were probably fascinated about the idea of emulation while most users claimed their interested was based on the idea of creating a (legal) backup copy and protect the original disc. One of the tools that came up during that time was, e.g., CloneCD[52]. It allowed users to backup an original CD and store it in a file on the hard disk. This file can be used to burn the copy on another CD, if the pirate owned the correct hardware. It also featured a Virtual-CD drive to mount such a copy into the system. Later on, it also provided features for DVDs and Blu-rays. In all cases, requirements by the copy protections were emulated by this software. The application recognized it as a real, physically present, and original CD [179].

Cracking internet-related games

As previously presented at the end of chapter 8.3, software today is most often distributed across the Internet and in online stores, and someone can still even buy games in usual stores on, e.g., CDs or DVDs.

For instance, games might be acquired from Steam[53] by registering and buying the games for an account. Many of these games require a permanent internet connection with a producer’s server, since they are often multiplayer-based and require interaction with other players worldwide. Nevertheless, even popular server-based games cannot be wholly protected by this approach, as crackers[54] may just start to emulate the whole server-infrastructure to allow pirated games to be played within a limited environment (e.g., World of Warcraft’s server emulation, also known as “private servers” [180] like the (recently shutdown) “Nostalrius” [181] servers with ~800.000 users [181]). Of course, these (illegal) private servers do not represent the actual game and lack many features that the producers add to a game over time, but it essentially allows the free gaming of a paid game/service. In general, these servers are usually illegally implemented by reengineering and sniffing the network traffic. They emulate the real server

[52] The tool is not permitted in Germany anymore - http://www.slysoft.com/de/clonecd.html
[53] http://store.steampowered.com/
[54] Someone, who cracks apps or games to get illegal copies working

step-by-step. Therefore, it can already be noted that a secure network traffic encryption also is essential.

Of course, even nowadays, cracks are still available to separate games from the requirement of such a mandatory platform. Nevertheless, this mainly applies to single-player games sold on these platforms, since they do not require a permanent server connection to function properly.

8.4.2 Cracking methods on mobiles

As stated earlier, Android apps also combine some of the former ideas for protection as well as for cracking them. It is essential to know that many games are meant for single players and therefore the interaction with others or a permanent connection is not required nor desired (cf. limited data plan), even this starts to change slowly now (cf. Pokemon Go app) and carriers provide special data plans [182].

Most often a user has some sort of market account (e.g., Google Play or Amazon AppStore account) and downloads purchased or free apps to his device that get associated with the account. Therefore, apps are most often related to the used user account (if received from a store) and may be installed on other Android devices with the same account. This information is stored on the server-side only, and usually not embedded into the app itself. Copying the APK file to a different device that has a different account will certainly trigger the copyright protection, and prevent the application from execution (when it is a protected app), but the actual copying is possible and not prevented by Android.

Nevertheless, apps might be easily cracked with the appropriated tools or by manually performing the required tasks. One of these tools is, e.g., “Lucky Patcher” [49] that acts as a general cracking tool and gained extreme popularity among mobile app pirates. It may perform various actions against an app itself and the used services on a device (e.g., to disable signature verification and to circumvent license verification [64, p. 60]). A detailed analysis of this tool is presented in 10.1.3.

Also, our research revealed that the often used LVL is vulnerable against an MITM attack on rooted devices to intercept the communication and exchange license parameters as well as used signatures on the fly [64, p. 54ff] (see 10.1.5). Apps protected by Amazons DRM may even be cracked more easily, by only removing certain lines of code [64, p. 30ff] as outlined in more detail in 10.1.4.

8.5 Data protection and available soft- and hardware solutions

As outlined below, we see that data protection, either for privacy or security reasons, is always an important and difficult task and many factors need to be considered, depending on the actual way it is implemented and the preferred security level.

For instance, it might be more than sufficient [183] for regular users to encrypt their data at 128bits[55], while governmental agencies certainly need to use 256bits or even higher (cf. “TOP SECRET” requirement) [184]. While a thief will most certainly be unable to decrypt the user’s data encrypted at, e.g., 96bits, a foreign state with access to high computational power might crack it within days to a few months. Nevertheless, there might be a performance benefit for that user when using a smaller key size, and the required security level has to be weighted carefully depending on the desired needs.

In terms of mobile development and its copyright protection a good performance by the applications is the most desired goal. Any implemented methods should not affect the user’s app experience. Upon the start of an application, a few seconds delay is probably acceptable by most users, but any annoying disruptions during the runtime should be avoided at all costs. That becomes important when proposing, e.g., the usage of fairly slow SEs by us in the proposed solution section.

Due to the fact that Android devices may be rooted either by exploit or a predefined way by the manufacturer (cf. 6.3.1), data stored on the phone or exchanged over a network is not secured from eavesdropping, illegal access by its user (important in terms of copy protection), or by a trojan (in terms of privacy).

Any Android application stores its data (including settings, files and SQL databases) in the corresponding directory referred to its package name, e.g., /data/data/de.tum.nilsapp.

All data stored here belong to the corresponding user/group ID of that application with the exception of native libraries that are assigned to the system user. Nevertheless, a user with root permissions may access any files here. Thinking about any network activity, a root user may also intercept the network traffic.

Therefore, a basic encryption of files, databases and traffic is essential for simple protection already today. While any files may be encrypted using the typical Java Crypto or OpenSSL APIs [185], SQLite features the usage of extensions for encryption, too. A possible solution might be SQLcipher[56]. By default, there are secure versions of all major network protocols available (e.g., HTTPS instead of HTTP) and should be used whenever possible.

Ultimately, simple protection does not shield against sophisticated attacks such as interception of functions calls (see 10.1.5), but raises the time barrier until there is a breach of protection that protects the user’s data against theft (cf. Trojan).

In addition, use of obfuscation techniques is recommended that exist for Java and other languages to remove, e.g., references that would allow an easier reengineering of an application. A default tool available for Android is ProGuard [161]. It is shipped with Android Studio and provides basic protection already.

[55] Key length and a factor to describe the assumed security level
[56] https://www.zetetic.net/sqlcipher/

8.5.1 Secure Elements

General information

In general, SEs are available “in form of UICCs[57], commonly known as SIM cards, as an external flash memory card or even already embedded in the hardware of the phone itself” [186] (based on [187]).

Even Secure elements (SEs) [186] [187] are similar to smartcards, they have a “deutlich komplexeren Lebenszyklus […] was jedoch auch zu deren Flexibilität beiträgt“[58] [187] and may be changed dynamically by exchanging the installed applets[59]. The applets are created by using a special IDE (e.g., JCS Suite by Giesecke and Devrient) and developed using Java language.

Unfortunately, the internal SE as well as the SIM cards are of no practical interest to usual developers, and would require cooperation with carriers like Telekom, Vodafone, etc., or huge companies like Google, which is obviously very unlikely to happen for a smaller company. In addition, it appears Google does not want to support the UICC to be used as a SE due to disconnected SWP lines[60] between the NFC chip and the UICC [188], while the access codes to load applets to the internal one are unknown as well [189].

An alternative are SEs in form of external devices or to be used in an internal slot. There are SEs available by lots of manufacturers with each one featuring different options and feature sets or sizes. Besides different versions for the used card operating system, each card may support different cryptographic algorithms or holds different certifications to prove its security. Examples might be the “IDPrime MD” [190] by Gemalto, the “PS-100u SE” [191] by SwissBit or the “Mobile Security Card” [192] by Giesecke & Devrient.

Dissertation related decisions and options

In this dissertation the Mobile Security Card (MSC), an SD card with an embedded SE, by (formerly) Giesecke & Devrient Secure Flash Solutions was used mainly due to existing cooperation and freely[61] available tools (see Figure 35).

Figure 35 - Mobile Security Card [193]

Its parent company decided to shut down the Secure Flash division and discontinued the support for its cards. However, we decided to continue usage of the MSC, since it is used for

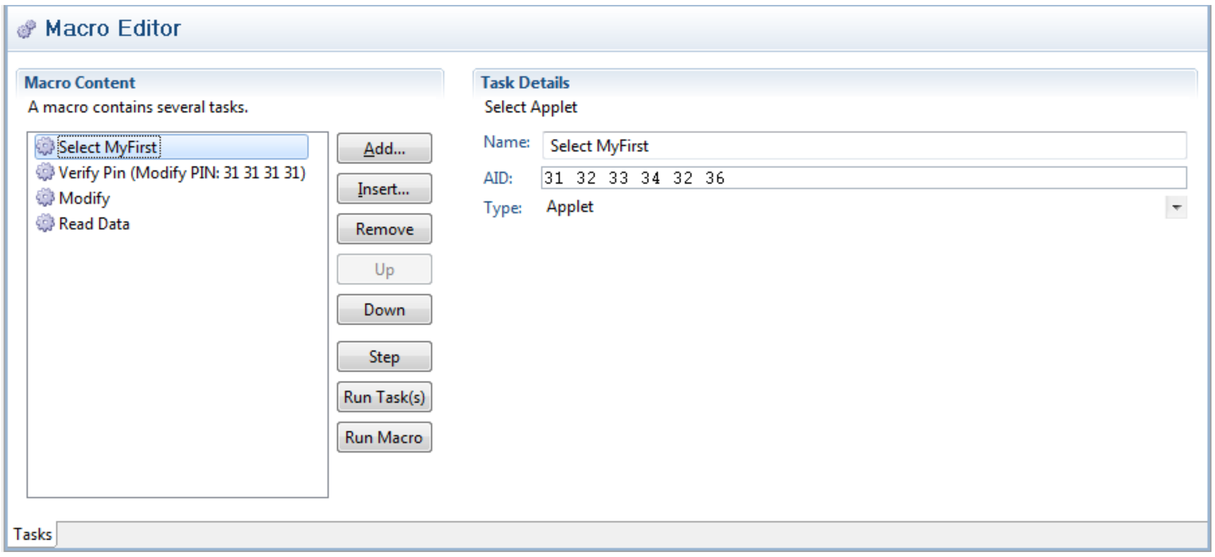
[57] Universal Integrated Circuit Card
[58] Translation by author: more complex life cycle […] [and offer] more flexibility
[59] programs on a SE
[60] Single Wire Protocol connecting a secure element and NFC modem [357]
[61] Sponsored by G&D SFS

demonstration purposes only. Especially, due to the fact that similar products became available that may be used with our presented methods in the future, e.g., the ones by SwissBit.

Development Tools

The IDE used for development of applets by Giesecke & Devrient consists of a modified Eclipse version featuring additional tools like an emulator of the SEs as well as a Macro Editor to verify and test the developed applets. Figure 36 shows the Macro Editor for testing the created applets and, e.g., selecting the applet with the AID[62] 31 32 33 34 32 36 as the first step.

In addition, and for debugging purposes, the communication between a (here: simulated) SE and an applet may be monitored and measured as shown in Figure 37. An interesting figure is the used time in milliseconds as it allows the assumption that the performance is quite low as outlined in more detail at the end of this chapter using a real device.



Macro Editor

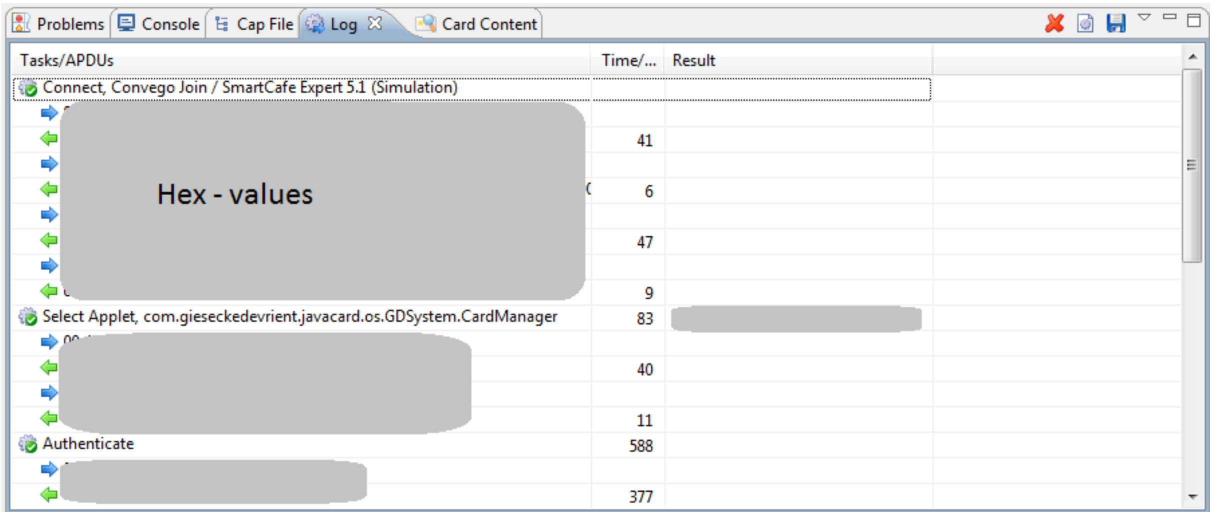
Macro Content
A macro contains several tasks.

- Select MyFirst
- Verify Pin (Modify PIN: 31 31 31 31)
- Modify
- Read Data

Task Details
Select Applet

Name:	Select MyFirst
AID:	31 32 33 34 32 36
Type:	Applet

Figure 36 - Giesecke & Devrient JCS Suite's Macro Editor



Tasks/APDUs	Time/...	Result
Connect, Convego Join / SmartCafe Expert 5.1 (Simulation)		
[redacted] Hex - values		
	41	
	6	
	47	
	9	
Select Applet, com.gieseckedevrient.javacard.os.GDSystem.CardManager	83	[redacted]
[redacted]		
	40	
	11	
Authenticate	588	
[redacted]		
	377	

Figure 37 - Giesecke & Devrient JCS Suite's Communication Log (without details due to NDA)

[62] Application Identification Number

Architecture of the Mobile Security Card

Internally, the MSC consists as shown in [194] and Figure 39 of a typical flash controller, the flash memory as well as the SE. It features the Java Card operating system and therefore, the dynamic installation of Java Card applets that run on top of a Java Card Virtual Machine (JCVM) within the Java Card Runtime Environment (JCRE). This includes management for memory, applets, and security as well. It provides a Java Card API for developers. The lifetime of the JCVM equals the lifespan of the card itself and any information is preserved upon power failure. Figure 38 shows the typical architecture of Java Card OS.

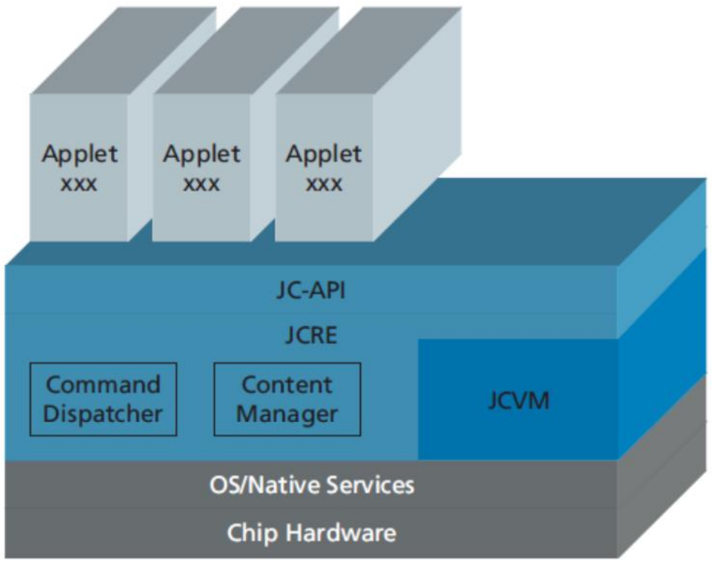


Figure 38 - Architecture of the Java Card OS [194]

The flash memory is available to any connected device by default methods (e.g., by mounting the device within the filesystem of the host computer). An important fact is that access by the SE to the flash memory is not possible [195], which reduces the possible functionality enormously, since data is limited to the provided internal memory of the SE of 78KB [192]. Also, the access to the secure element is limited, and may be established by using the ASSD[63] interface or by using the “Generic Security Interface (GSI)” [194] that uses a usual file I/O operation (special file) for the communication of an app with the secure element [194].

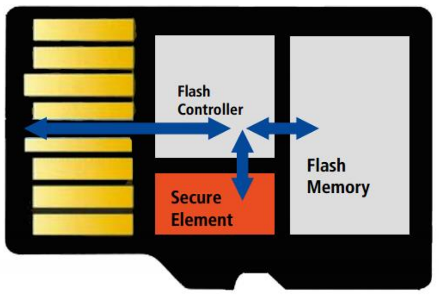


Figure 39 - Internal Architecture of the MSC [194]

[63] Advanced Security SD - Specified by the SD Associations [194]

Card Management and Security configuration

The actual application management (dynamic updating of applets) is of no importance here and was not used in the solution proposal, since the existing framework was not compatible with modern phones (see next section) during the practical phase of this research work. Therefore, features had to be left out, and that feature was unimportant within our demonstrator solution. However, we assumed that the card is used by a single company only, which is relevant in terms of security (see 10.4.1 for details).

By default, the so-called “Issuer Security Domain” [196, p. 39ff] is responsible for managing the keys and delegating permission to others, e.g., to modify the card content by installing another applet.

In general, the MSC also “complies with the Global Platform” [194] standard that fills the gaps by the Java Card standard. The Global Platform standard defines default methods like the requirement of an “Issuer Security Domain” [196, p. 40] and secured channels for the management.

Communication with the Secure Element (applet)

The communication between the flash controller (and external requests) and the Secure Element is specified by ISO7816[64]. All requests are encapsulated in an APDU, an “Application Protocol Data Unit” [194] that may be interpreted on the card by the process method (see Figure 40 for a basic example of such an applet) [194].

```
public class MyApplet extends Applet{
// defining constants and memory allocations
...
public MyApplet(byte[] bArray, short bOffset, byte bLength){ // constructor
register(...);
}

// install creates a new MyApplet instance which will be registered with its AID
public static void install(byte[] bArray, short bOffset, byte bLength){
new MyApplet(...);
}

public void process(APDU apdu){ // processes incoming commands
switch (instruction){ // forwards incoming instructions to the defined methods
  case command1: ...
  case command2: ...
  default: ...
  }
}
// private methods
...
```

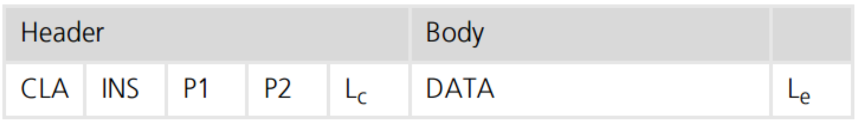
Figure 40 - Example Code of the default applet structure [197, p. 22]

APDUs contain instructions that may initiate different methods on the card besides having parameters. In the current implementation each APDU may contain data of up to 255 bytes [197].

Figure 41 and Figure 42 illustrate the default possibilities for the message structure of these APDU requests. The different fields are defined as follows [196, p. 158]:

[64] International Standard, cf. http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816.aspx

CLA	The bits of CLA define the command type (b8 to b5), b4/b3 for secure messaging indication and b2/b1 represent the used logical channel (see ISO 7816-4 for details).
INS	This byte represents the instruction byte and may be defined by the developer in its applet.
P1/P2	Parameter bytes
Lc	“Length of command data” [196, p. 158]
Le	“Length of expected response” [196, p. 158]
Data	Payload



Header					Body	
CLA	INS	P1	P2	L_c	DATA	L_e

Figure 41 - APDU default structure [196, p. 158]

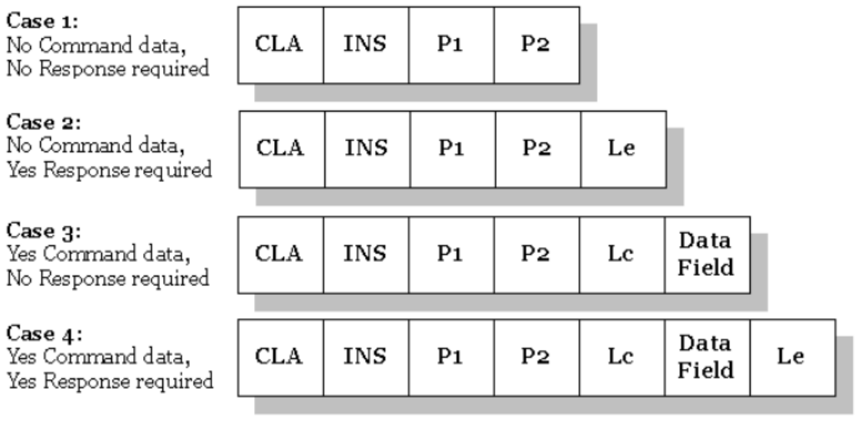


Figure 42 - Format of APDU requests [197, p. 25]

Issues on modern phones and new USB-OTG requirement

Since most modern phones (e.g., Nexus 4, Nexus 5, etc.) [186] [164] do not feature an SD card slot anymore, any solutions are already limited to the GSI interface from the beginning. The solution to access the MSC on modern phones (before Android Lollipop, see next section) is, e.g., the usage of a Micro-SD-Micro-USB-Adapter using USB-OTG. USB-OTG is a technology to support the host-mode on smartphones or other devices for connecting peripherals such as, e.g., an USB flash drive [198]. As evaluated by a student group, Figure 43 shows an overview on major devices that support (green), partially support (yellow), or do not support (red) USB-OTG. Those devices marked red will not be able to use any of our presented solutions.

One additional issue is that mounting any external device requires root rights on many devices, and most smartphone users would not be able to fulfill this requirement. In addition, we discovered that some Android versions do not support the O_DIRECT flag, which allows unbuffered read/write access – a critical requirement to read the reply to a command. As a possibility to address the issue of mounting the MSC without root rights (see [199] for technical details on the following library) and surrounding the O_DIRECT problem that affects some Android versions, the library libaums[65] was developed and later extended by MSC capabilities to be used in this research project. A disadvantage of this approach is the raw communication with the SE, which is usually handled by the appropriated framework, e.g., the "MSC Smartcard Service" [200] by Giesecke & Devrient. For this reason and to increase the security once more, it was decided to define our own protocols for the communication between an applet and an Android application, asides from the usage of some default, standardized commands (e.g., for selecting an applet).

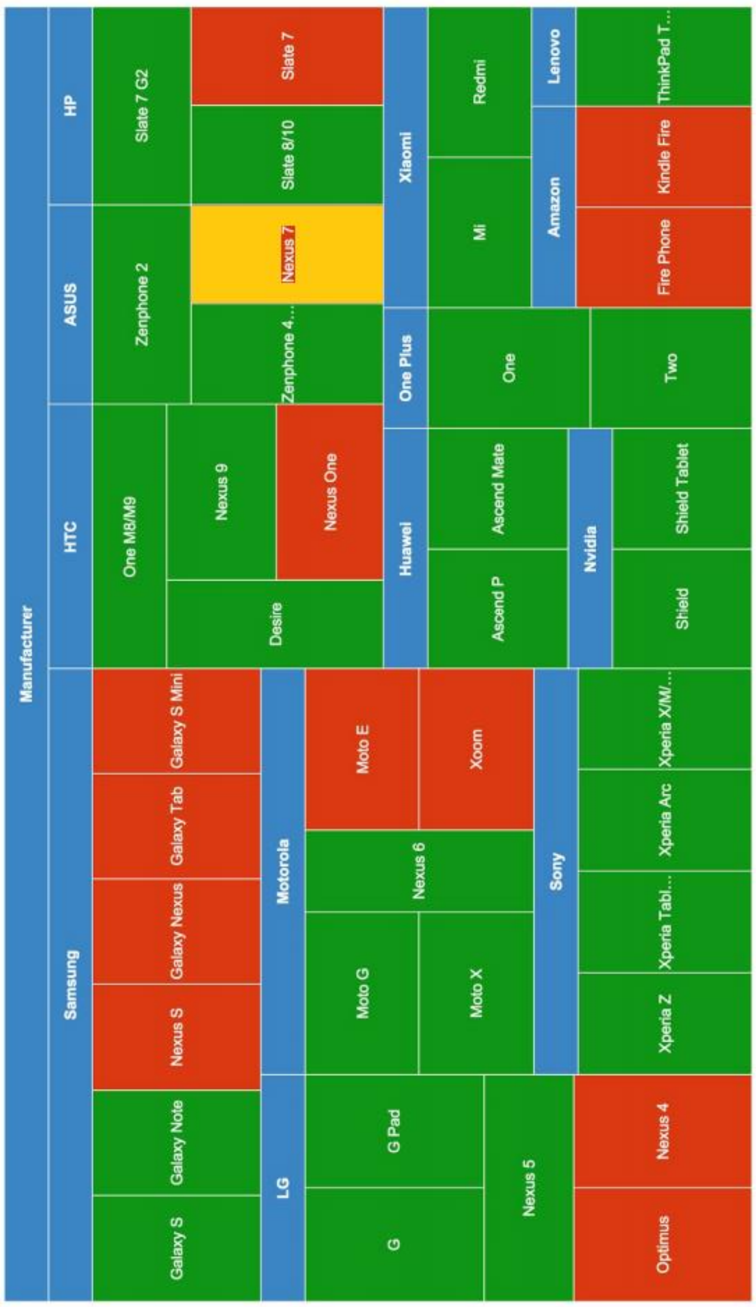


Figure 43 - Devices supporting USB-OTG (as of 2015) [201]

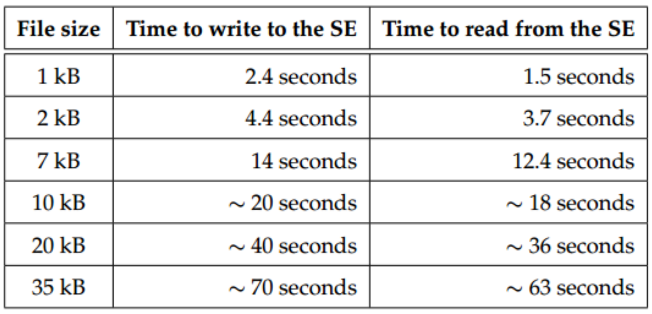
[65] "Library to access USB Mass Storage devices", M. Jahnen, https://github.com/mjdev/libaums

Issues on modern phones using Android Lollipop or higher

While USB-OTG is a mandatory requirement, using native code is an additional, mandatory requirement in terms of gaining additional security (see 10.3ff). By using a native port of the aforementioned libaums library, we noticed that it is not working on Android versions higher than or equal to Lollipop. Further details and remaining options are found in 11.4.7.

Performance of Secure Elements

Another important fact [197] about SEs is the weak performance. A typical I/O request to the secure element performing a request and receiving a reply takes about 200ms with a payload of 255 bytes (with our implementation). We evaluated the results of a student's thesis and found (as shown in Figure 44) that a file size of 1kB generated data in Android, was already transferred and stored within the SE in 2.4 seconds. Reading the same data from the SE back to Android took additional 1.5 seconds. This low performance limits the usage of SEs dramatically and we cannot justify more than a few seconds in terms of a copy protection in general. In fact, it is preferred that the impact of any methods is not detectable by a user (< 2 seconds).



File size	Time to write to the SE	Time to read from the SE
1 kB	2.4 seconds	1.5 seconds
2 kB	4.4 seconds	3.7 seconds
7 kB	14 seconds	12.4 seconds
10 kB	~ 20 seconds	~ 18 seconds
20 kB	~ 40 seconds	~ 36 seconds
35 kB	~ 70 seconds	~ 63 seconds

Figure 44 -Performance test of the MSC using libaums in an Android App (10kB and more were calculated) [197]

8.5.2 Trusted Execution Environments

As previously introduced in the related work section (see 9.1), Trusted Execution Environments (TEEs) may provide a secure setting for confidential app data by separating the exploitable operating system (Android) from a 2nd secured operating system on a device. Advanced TEEs, like Samsung's Trustonic for KNOX, may even provide dedicated hardware access to allow a secured interaction with a user [202]. A simplified figure explaining an Android device that provides a TEE is displayed in Figure 45.

As clarified by Mr. Ekberg[66] in [203] for the Trustonic TEE, their solution also separates the TEE OS into user-space and privileged-space. By default, that side is assumed to be secure and each TA (Trusted Application) acts within its own (user) address space within the TEE OS. Typical use cases for TEE in general might be, e.g., mobile payment, BYOD, secure hardware tokens, runtime integrity verification, but also DRM including HDCP and similar protections [204, p. 7].

[66] Employee of Trustonic

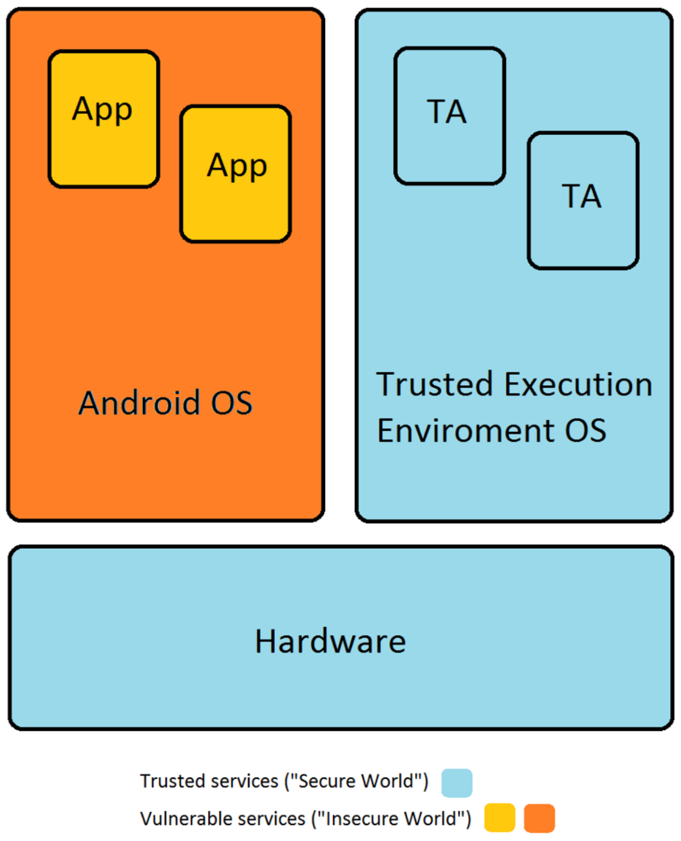

App
App
Android OS
TA
TA
Trusted Execution Enviroment OS
Hardware
Trusted services ("Secure World")
Vulnerable services ("Insecure World")

Figure 45 - Simplified overview of an Android device providing a TEE (based on [203])

Nevertheless, there are [205] several TEEs options available (e.g., QSEE, HTC's modified one or the already mentioned solution by Trustonic), and it seems each manufacturer is creating its own (cf. "ARM TrustZone® is one way […] [it is] not the only way" [204]).

Therefore, and similar to Android, TEEs, for now, face a similar fragmentation based on the used chipset. It may even happen that two TEEs share a SoC[67], e.g., QSEE and Trustonic [206]. Similar to SEs, the Global Platform standard also defines default methods for TEEs [206]. Current systems "build on trust […] that you as an attacker should not be able to do anything in that world [...] [and that it is] unreachable"[68] [205]. Most TEEs come as integrated hardware solutions within a SoC, e.g., "Qualcomm's Secure Execution Environment" ("QSEE") [205] on a SnapDragon processor.

8.5.3 Enhanced Operating Systems

For the sake of completeness another option for increasing data protection is to harden the system against possible exploits by limiting the access to system files or other relevant files that do not require permanent accessibility by a user.

For instance, the NSA, along with SEAndroid (originally SELinux), developed an improved system with mandatory access control that was later integrated into Android itself [103, p. 12]

[67] System on a Chip
[68] Transcript by author

[207]. It is also responsible for limiting the access to SEs in later Android versions due its access limitations for device files. This restriction deeply affects our ideas of using SEs with native libraries as outlined in 11.4.7 in more detail. While meant as an additional security measurement, it has the very opposite effect in this special case.

Moreover, a similar approach to secure Android, was presented by Sven Bugiel et al. with their modified Androidsystem called “TrustDroid” [208]. Moreover, the solution approach by TUM I20 and FORSEC (“TP1: Security Architecture for Mobile Devices” [209]) is another solution in that category.

Further details on these approaches are not of importance, since hardening Android itself is out of scope of this dissertation and not considered the correct way for copyright protection due to Android’s large size with many vulnerabilities in several services and by a large number of developers (cf. common saying “too many cooks spoil the broth”). From the author’s perspective it is difficult to secure Android without hardware modifications and recently discovered exploits with regard to Qualcomm chipsets and their drivers perfectly confirm that assumption affecting millions of devices once again [27].

9 Related work and discussion

Due to its openness Android is also a target for security researchers and hackers of all kinds. While this dissertation is about copyright protection by using e.g. native code and secure elements, other researchers in related areas focus on copyright protection mechanism in hardware or software, general security or privacy issues and possible solutions instead.

The following section should present an overview and short introductions on recent works in related areas as well as a comment about the relations to this work and/or issues. Moreover, the last section provides a short comparison to my proposed solutions.

Any theses by my students are not included here, since they belong to our research group.

9.1 Securing and protecting Data

For instance, in their paper **Tim Cooijmans et al**. [210] analyzed secure key storage solutions and confirmed that keys might be securely stored depending on the used solution (cf. ARM's TrustZone vs. Bounty Castle[69]), which are non-accessible to root attacker(s), and even those intercepting communication. Ultimately, however, they may still be used by the attackers as by any legitimate app. Depending on the device, keys might be secured and perhaps not viewable, but can still be used illegally by faking legitimate app requests.

Comment: Even if the keys are stored securely, the issue of an insecure Android world is clearly visible.

On the **Google IO in 2015, Peiter Zatko** [211] introduced the community to "Project Vault" [211] after Google realized that customers do not yet have their own secure elements. For the moment they focus on enterprising customers first and use it internally (e.g., "Project Abacus" [211]). "Project Vault" [211] is a microSD card that features an ARM processor running an RTOS (realtime operating system). Therefore, it can act independently from the host OS. Also it includes an NFC chip and an antenna, which are used for authorization purposes. Furthermore, it supports various security features for "hashing, signing, bulk encryption, streaming encryption, a strong hardware random number generator and four gigabytes of isolated sealed storage"[70] [211]. For the moment the communication works by using two files. One for sending requests and one for receiving information. Everything is still experimental and Google released "Research Hardware […] [and a] Development Kit" [211] only (status in 2015). According to Spiegel [212] it should be also used to make passwords unnecessary.

A quite similar product is the "FIDO U2F Security Key" [213] by Yubico; it uses a secure element internally (cf. [213]'s FAQ). It may be used to authenticate against various services including Gmail or any other website supporting the FIDO U2F protocol.

Comment: From a technological point of view, "Project Vault" [211] and the "FIDO U2F Security Key" [213] are each close matches in terms of related works available, even though

[69] Software solution
[70] video transcript by author (starting ca. 52:57)

they are obviously not focused on copy protection yet. Nevertheless, if they would be available for a broad range of devices and customers, it may allow for the future usage of current, conceptual, and proposed methods that cannot be used due to USB access issues caused by SEAndroid (see 11.4.7 for details).

Another topic of interest is Trusted Computing. Usually trust is generated by using a hardware module “known as […] Trusted Platform Module (TPM)” [[214] as cited in [215]]. While TPMs are “dedicated microprocessors designed to secure hardware” [216, p. 5], TEE “is a separated execution environment that runs alongside the Rich OS […] [and provides] isolated access to its hardware” [216, p. 4] instead. **Trustonic** is one of the leading companies providing TEE [217] that derived from existing products like **ARM’s TrustZone, G&D’s Mobicore** and other vendors [218]. Even “mobile phones with hardware-based TEEs appeared almost a decade ago, and today almost every smartphone” [219] includes one, “the use of TEE functionality has been largely restricted […] [and there] has been no widely available means for application developers” [219]. With “Trusty” [205] Google is working on its own TEE. However, on its website it is still declared “subject to change” [220] and appears unfinished.

Comment: As highlighted in the security analysis chapter for hardware (see 10.4.2), TEEs are introduced to the market, but still require improvements and are not the Holy Grail for solving all kinds of security issues. While performing research for this dissertation, it was not possible to obtain developer access for hardware and software in a reasonable amount of time, but based on the results by others (see 10.4.2), we believe its market introduction is still ongoing and various research is and has to be performed before a standard solution becomes available. First, that solution might be the base for research of copyright protection using TEEs.

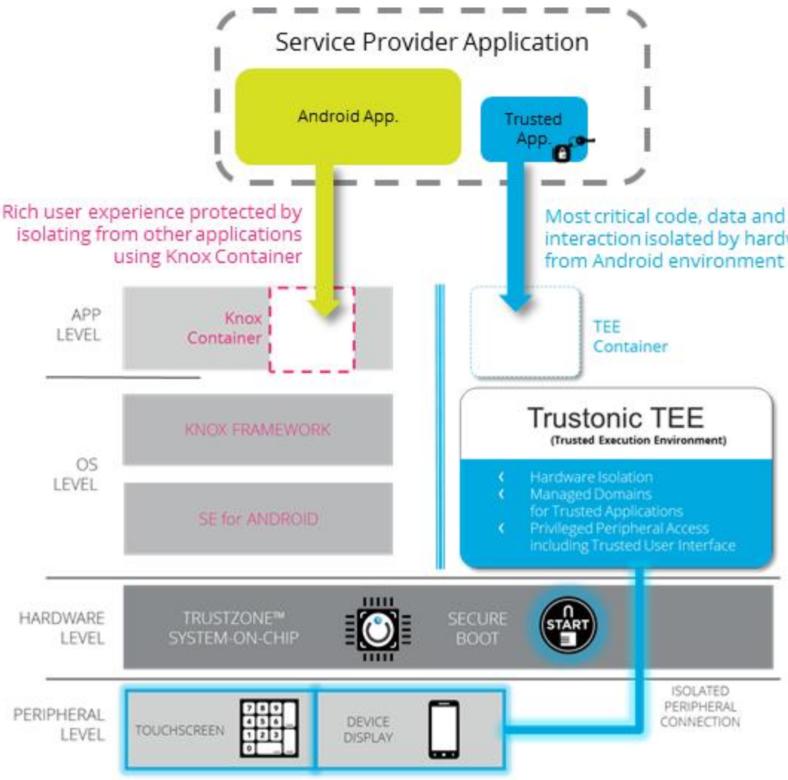

Service Provider Application
Android App.
Trusted App.
Rich user experience protected by isolating from other applications using Knox Container
Most critical code, data and user interaction isolated by hardware from Android environment
APP LEVEL
Knox Container
TEE Container
OS LEVEL
KNOX FRAMEWORK
SE for ANDROID
Trustonic TEE
(Trusted Execution Environment)
Hardware Isolation
Managed Domains for Trusted Applications
Privileged Peripheral Access including Trusted User Interface
HARDWARE LEVEL
TRUSTZONE™ SYSTEM-ON-CHIP
SECURE BOOT
START
PERIPHERAL LEVEL
TOUCHSCREEN
DEVICE DISPLAY
ISOLATED PERIPHERAL CONNECTION

Figure 46 - Trustonic for Samsung KNOX [202]

In addition, **Samsung** developed “Samsung KNOX” [221] that integrates (enhanced) security solutions like TrustZone (“TIMA”[71]) and SEAndroid, while also featuring, e.g., secure boot capabilities and container solutions. Furthermore, it is meant for governmental and enterprise usage due to its management features and certifications [221]. The name is related to the famous Fort Knox [222], which is known to be one of the most protected facilities worldwide.

An even more advanced TEE version **by Trustonic** is “Trustonic for Samsung KNOX” [202], since it adds isolated access to the display and touchscreen for entering credentials in a secure manner and is separated from the insecurity of the Android OS. Figure 46 illustrates the architecture in a diagram.

Comment: Secure access between the user (touchscreen/keyboard) and the secured world is an essential advantage of Trustonic’s solution, while TEE may greatly improve the security in terms of copy protection in the future. As outlined before, it would be necessary to define the standard available on all Android devices. An outlook for such a solution using an Android-based TEE is given in 11.2.2. Of course, this cannot apply to older (existing) systems, which is the goal in our research approach.

In [223] **Luca Flasina et al.** presented a secured DexClassLoader[72] library called “Grab ’n Run” [223] that allows, e.g., the dynamic loading of remote code into the current program in a more secure way by wrapping the existing DexClassLoader provided by Android (Google), while also verifying integrity and signatures.

Comment: The library and research results are of interest to developers trying to dynamically load code in a more secure way, so to prevent malicious injections (cf. MITM attack) and in terms of data privacy maybe. In terms of copy protection their approach would be of interest in a native code version instead, since it is more protected against reengineering (see 10.3ff), while allowing the loading of additional program parts upon successful license verification.

9.1.1 Stealth techniques

In addition, all topics related to malware such as those in the paper introduced by **Thansis Petsas et al.** [224] are of interest, since malware-methods may hide our protection techniques against analyses by attackers. For instance, their work introduced the issues of “dynamic analysis of […] malware” [224] and possible techniques to prevent analyzes. For example, many emulators intend to have unrealistic “values for static properties like the serial number [or] […] [outputs of] the accelerometer” [224] and other sensors. These methods can be used to prevent analysis and even the use of different opcodes, as presented in the paper by **Wu Zhou et al.** [162] might complete stealth technologies, while helping to block the usage of certain reengineering apps completely [224].

Comment: While the identification of emulators or dynamic analysis tools sounds reasonable, the more than interesting approach by Wu Zhou et al. is no longer useable on newer Android

[71] TrustZone-based Integrity Measurement Architecture for monitoring the Linux Kernel [221]
[72] The DexClassLoader class allows the dynamic loading of application code from APK/JAR files [223]. See 11.4.3 for further details.

versions due to ART's pre-compilation. It can still be used as a type of encryption and for loading libraries.

In general, all related research work for recognizing reengineered and repackaged apps is also interesting for a copyright protection. **Hugo Gonzales et al.** [225] discovered the so called "String Offset Order" to identify repackaged apps. This affects "the data section of the .dex file" [225] and its included, "string identifiers list" [225], where strings are arranged in alphabetical order. Tools for repackaging use a different method and therefore, may be discovered.

Comment: The possibility to discover manipulations is interesting in general and the original APK file used for the installation, is stored as base.apk by Android in unmodified[73] form (see 7.3.12 for details). Nevertheless, when a repackaging takes place, it needs to be pointed out that while signing the APK with a different key, the signature will be different anyway.

Besides system modifications and –hardening, an interesting approach by **Daniel Hugenroth et al.** [226] is the "Obfuscation using Self-Modifying Code" [226] in Android apps themselves. Here the code is modified during runtime to allow the execution in the right manner, while the decompiled code leads the attackers to false assumptions.

Comment: The dynamic manipulation of variables is an interesting approach that we try to improve by using secure elements and native code in a slightly different way in the proposed solution section of this work.

In [227] **Patrick Colp et al.** introduce methods to store data on SoCs instead of DRAM to prevent memory attacks, while unencrypted data is not ever stored in DRAM. They are using mechanisms intended for embedded systems originally and that are ARM-specific. According to their paper, they are still available on mobile devices.

Comment: While their paper provides interesting information on all kinds of hardware attacks, their solution might be of interest to hide information (e.g. encryption keys) even better, while typical attackers may probably dump the main memory only. It certainly extends their research time and most developers (attackers) do certainly not include the processor cache in their thoughts.

9.1.2 Exploit prevention and access control

Despite these trusted computing methods developers tried to harden Android by adding several security features in former days. A famous approach that got included into Android was developed **by the NSA** called "Security Enhanced (SE) Android" [103, p. 12] (or short SEAndroid) and includes several improvements like Mandatory Access Control and the possible avoidance of privilege escalations. In addition, other researchers including **Sven**

[73] verified with an app by July 10th 2016

Bugiel et al. [208] implemented a modified Android system called "TrustDroid" [208] that for example enforced "mandatory access control on the file system and on Inter-Process Communication (IPC) channels [as well as] […] the network layer" [208].

Comment: Software related solutions always risk exploits and provide all kinds of gateways for malicious issues that they cannot handle, e.g., driver issues by third parties that lead to root access (cf. "Quadrooter" [27]).

Furthermore other researchers and **companies like Zertisa** have the goal to separate private and commercial data or apps by introducing virtual machine concepts to Android devices [228].

Moreover, **FORSEC, in cooperation with TUM I20,** introduced in a poster [209] the "TP1: Security Architecture for Mobile Devices" [209] that is based on TrustZone and XEN virtualization. They have the goals to provide, e.g., "Secure, reliable multi-tiered architecture for hand-held and mobile devices" [209], while performing an "Evaluation of machine learning approaches for automated incident detection and response" [209] in addition to other goals.

Comment: These approaches require a rooted smartphone or one that already includes the required firmware by default. Instead in our research, we try to avoid this and the solution should be usable on stock Android devices (primarily phones/tablets). Moreover, the research work is more related to protecting data from malware than dealing with license issue. Even the same security measures may protect it.

9.2 Copyright protection

9.2.1 By smart cards or similar devices

More than a decade ago **Thomas Aura et al.** [1] already worked on the topic of smartcards in combination with licenses. Their paper describes methods to use smartcards for storing licenses and their secure distribution to other smartcards by using private-public key mechanisms, while maintaining the license goals (e.g., one license for each copy). Even a decade ago these researchers summarized that "there are always ways to work around the protection mechanisms [and only] […] the time to market for pirated copies [may be increased] and that pirated products cannot be sold as authentic" [1]. These statements still apply today and "copy-protection is always to some extent security by obscurity" [1]. Furthermore, they mentioned that reengineering of smart cards "must be too expensive or time-consuming" [1], while "modifying the software to run without the card […] must be equally difficult" [1].

Comment: Unfortunately, the statements made ten years ago are still valid today, and certainly will be forever; however it may be assumed to be very difficult to break security measurements of, e.g., a secure element.

Shoaib et al. [229] took an approach by using smart cards for storing key information, while encrypting the DEX file and supplying customers with an encrypted version. In addition, a license server was used. The decryption was performed in memory and used the method "private static int openDexFile(byte[] fileContents)" [229] method for later on loading the executable. Furthermore, assets and other resources are not protected.

Comment: Unfortunately, Google decided to remove the dynamic loading from memory in newer Android versions and therefore their approach is not possible anymore, without updating it (cf. discovered solutions in 11.4.3).

The **company Aktiv Soft JSC** [230] developed a "high-performance dongle [called e.g. Guardant Code] with built-in cryptographic algorithms [and] up to 384KB of memory to store loadable code" [230]. Depending on the version, it even features an RTC to allow timed license models. By presenting itself as an HID device, it does not require additional drivers and is available to most platforms, including Android. On Android platforms it requires a service that allows Android applications to interact with connected dongles, while app developers are provided with a Java API to use the dongle/service within their applications [231].

Comment: The presented solution is similar to available SE solutions like the MSC by G&D that require developers to use a service for interaction with the MSC also by default. Nevertheless, due to our cooperation, we circumvented this (insecure[74]) service-based access and provided direct connections between the hardware and the application by using native code for additional obfuscation up to the Android versions activating SEAndroid in enforcing mode, which have unresolvable issues at the moment (see 11.4.7 for details). Furthermore, one key difference is perhaps the much better performance of the Guardant Code dongle in comparison to the MSC. Ultimately their product would be of interest if they would provide native C versions of existing libraries, while Google or the manufacturers need to permit USB access for the Android NDK.

In general, smart cards are known to be used in various DRM solutions of PayTV distributors. For instance, "VideoGuard" [232] by Cisco Systems is one such example.

9.2.2 By additional virtualization

Wu Zhou et al. [162] also presented in their paper a very interesting approach by introducing "the first VM-base [sic!] protection system for Android" [162]. It works by transforming Dalvik Bytecode and its opcodes to a new format that results in rendering the known reengineering tools (e.g., baksmali[75] or dare[76]) useless, since they are unable to understand the unknown opcodes. In terms "DIVILAR […] hooks into Dalvik VM" [162] to execute the code in the correct manner again. According to their investigations the overall performance is not affected that much and on average, as low as 16,2% more overhead time.

[74] cf. interception with Xposed framework (details later)
[75] Disassembler - https://code.google.com/p/smali/
[76] Retargeting tool - http://siis.cse.psu.edu/dare/

Comment: Even that approach is very interesting, it is not usable on newer Android versions anymore. The same applies to obfuscation by using junk bytes that prevented reengineering tools from working [233].

9.2.3 To identify software piracy

Another approach by **Joohyouk Jang et el.** relates to apps themselves is called the "Steganography-based Software" [234] watermarking of Android applications for proving "the ownership of [an] […] application developer and [to] verify users who purchased and illegally distributed their copies" [234]. For accomplishing these goals, the app receives a watermark by the producer and each app-copy also includes user-specific watermarks. "The proposed scheme embeds watermarks by reordering the sequence of instructions in the basic blocks in Dalvik executable files" [234]. The watermarks are checked upon first installation or during its initial run, and the desired action can be executed [234]. **Hyunho Ji et al.** [235] describe a similar approach for detecting illegal apps by using fingerprinting technologies too.

Comment: The detection of modifications (Is the app cracked?) is of interest, but due to the optimizations and - by ART - compilation (see 7.3.8 for details), it is complicated to verify safe ways for these calculations. Common cracking tools like Lucky Patcher can work on both, the APK files (detectable) and the optimized versions (changed checksum anyway) instead [66] which are not covered by the method presented in the above papers. Their approach circumvents these issues and affects the original APK file that is still available on modern phones and even within the optimized compilation files created by ART VM (cf. OAT files / see 7.3.9). It may identify an initially cracked app, but it will not work with the mentioned hacked/optimized versions of the app on newer Android versions. In addition, it requires a different app market that applies its methods to the APK file.

9.2.4 By using encryption and server-based solutions

Papers on copyright protection mechanisms propose various ideas. For instance, **Sung Ryul Kim et al.** [236] recommend a combination of "Online Execution Class" [236] a technique that loads app parts from a server as soon as they are required, and "Encryption-based Copyright Protection" [236], which decrypts app content on the fly, when needed. Figure 47 below, taken from another paper [237] explains their approaches in more detail.

Comment: The encryption and obfuscation of local code is a common way of protection. Unfortunately, Google removed the possibility for loading DEX code from memory dynamically on newer Android versions and their approach is not possible anymore without storing an unencrypted version in a file or using native code instead as explained in [100] and within this work in 11.4.3.

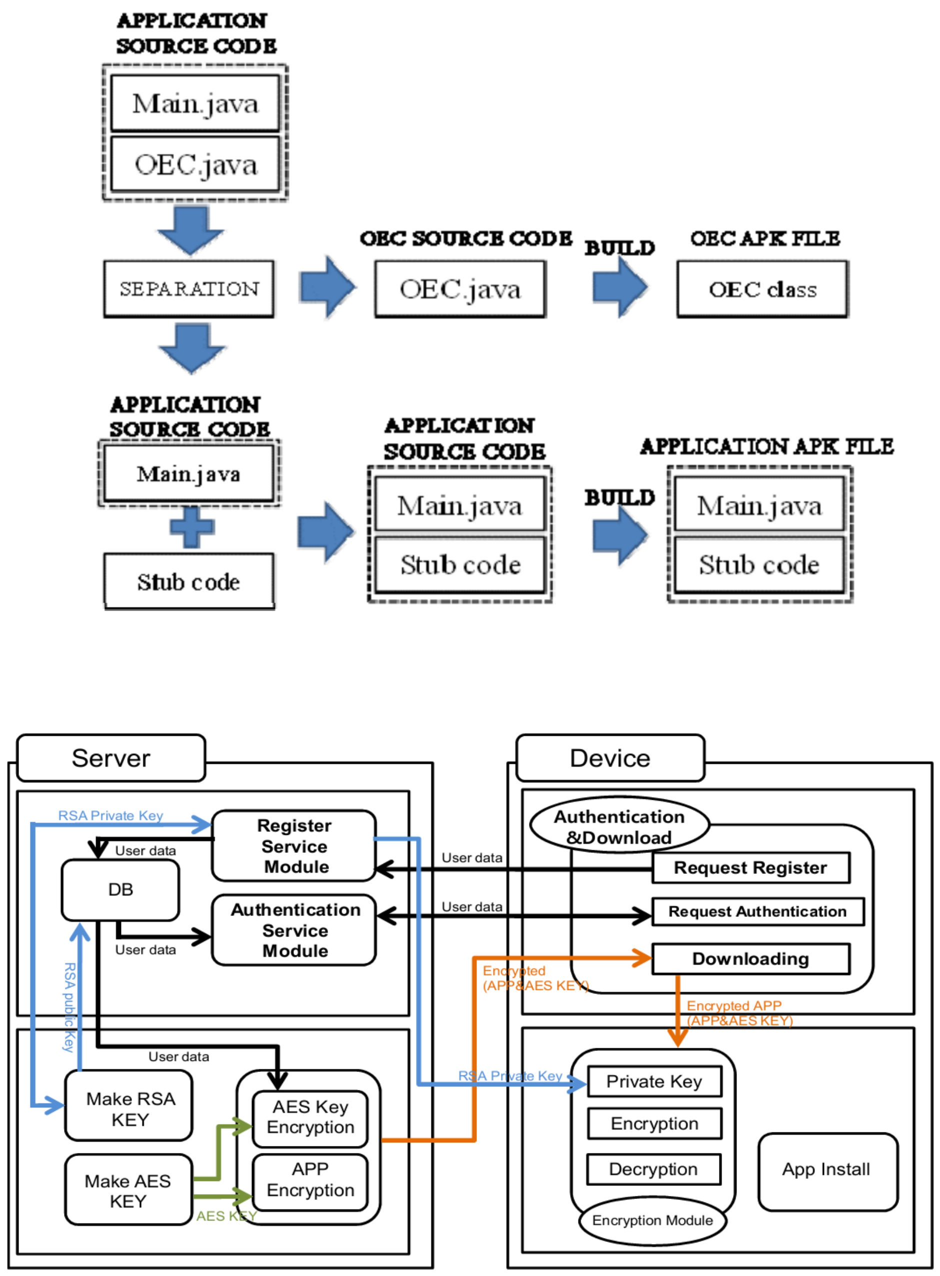


Figure 47 - "Online Execution Class" (top) and "Encryption-based Copyright Protection" (bottom) [237]

Youn-Sik Jeong et al. [238] presented a similar approach by dividing an app in an “Incomplete Main Application (IMA) and Separated Essential Class (SEC)” [238]. Here the first part is provided to users through markets, while the additionally required part becomes available after successful authentication against a market server. In addition, this part is stored locally in a secure space after the initial download. The secured space is created “by using a loadable kernel module (LKM)” [238] that hooks the calls “sys_open and sys_create” [238]. Now it checks for target process ids of permitted services and allows or declines the access.

Comment: The solution requires system modifications and/or root rights and therefore it is a very theoretical solution that cannot be used on any existing platforms. It is not of interest to our work, but surely an alternative solution.

Furthermore, **Kuo-Yu Tsai** [239] presented a copyright protection using a semi-trusted loader that receives encrypted and required program parts from an alternative market and upon first run, stores them encrypted and re-authenticates them upon each future run to receive required keys for the decryption again. Tsai claims that it is safe against a rooted device since users cannot use the APK file nor the encrypted files.

Comment: While the author’s claim seems to be true on first sight, it needs to be assumed that any keys can be intercepted on a rooted device using, e.g., the Xposed framework instead (= not safe on rooted devices), which can ultimately be used to build a fully decrypted app (see 10.1.5 for an example using Xposed). Also using Xposed, the app is not modified and internal check routines are not triggered. The author did not mention any special routines against memory attacks as performed by Xposed (see options in 11.5.2). An additional issue comes with the publication of the detailed method, and in theory, an attacker can follow up the provided guide (publication) to crack the protection; however the usage of Java code is not safe either. For that reason, we recommend customization and native code in our solution proposal (see 11.1.1). Also, it remains unclear how the author of the paper loads the code dynamically, since the Android versions available in 2015 require a decrypted file for loading code. It is not possible to load (decrypted) Java code from memory anymore (see 11.4.3 for details) and the paper’s author presumably addresses an old Android version.

9.2.5 By using a library

Google offers the license verification library (LVL) that needs to be integrated by the developer himself to check and act on the license response [47].

Instead **Amazon** integrates its Amazon DRM itself, when a developer publishes an app on the Amazon App Store [46].

Samsung provides an additional library for their smartphones called “Zirkonia” [240] that works similar to the LVL by Google and uses a native library, as well as a Java library, which needs to be implemented by the developer, to check the license and act accordingly [240].

In addition, **SlideMe** offers developers a similar way by letting them integrate a protection library to receive and act on the license replies [65].

Comment: The library solutions by SlideMe, Samsung and Google work in a similar way, and it is up to the developer to integrate them into the applications in a secure manner. Instead, Amazon handles the integration for the developer and may be more convenient. As described in the security analysis (see 10.1.4) it needs to be noted that these protection methods may be easily cracked (cf. [64] [75] [241]).

9.2.6 Used on x86 desktop computers recently

Another interesting copy protection available this year is a protection named after its company, the Denuvo copy protection. While there is little information in their FAQ available [242], it is apparently an anti-tamper protection ensuring that DRMs by Steam or Origin[77] are not bypassed affecting non-performance critical program functions only, while not constantly encrypting or decrypting data. It is used for games as “Star Wars Battlefront, Just Cause 3, and FIFA 16 [keeping them] piracy free for months” [242]. According to various sources [243] [244] it appears to be the pirates’ nightmare and is extremely difficult to crack. Nevertheless, as reported by [245] recently, also Denuvo was circumvented recently. On request Denuvo replied in an email [246] that the full protection is available for Windows only, while supporting a “lightweight” version based on Google’s LVL and anti-debugging features for Android only.

9.3 Protection against reengineering attacks

Besides available papers several default solutions exist to protect apps against reengineering. Default ones that should be mentioned for Java source code are ProGuard [161] and its improved, commercial version DexGuard. The last one provides several features including size- and performance improvements, name-, resource- and code protection as well as further features [160].

Also native C code using Android’s NDK with maximum optimization enabled, can be further protected by using an obfuscator like Obfuscator-LLVM. According to [247] Obfuscator-LLVM should yet not be used for production code and is still under testing. It supports various programming languages like C or C++ and works on all platforms supported by LLVM, e.g., ARM or x86, it is “working on the Intermediate Representation (IR)” [247] level.

9.4 Reengineering tools

Taking a look at the attacking side where **Collin Mulliner et al.** [248] described a similar approach to ours’ (see comment below) in hacking Google’s In-App-Billing by intercepting and replacing the target calls with their methods using their own library that “targets stock Android devices and does not rely on replacing core components” [248].

Comment: In comparison to our approach (see 10.1.5) for hacking Google’s LVL, the Xposed Framework was used for the interception/replacement and the method is slightly different in general, since we targeted the LVL and not In-App-Billing. They discovered as one of the issues that developers “solely [rely] on client-side enforcement […] [and confirm our impression] that many app developers […] [are not] aware of dynamic attacks” [248].

[77] Both sale platforms for games etc.

Ho Kwon Lee et al. [237] took a similar, but more generic approach in their paper by analyzing the possibilities when watching the main memory. They discovered possible issues, since the "App contents can be read off the main memory […] [and advised that] copyright protection techniques must be enhanced to include this possibility" [237].

Comment: The general issue that Android belongs to the insecure world is a known fact. Therefore, we can already conclude that it is very hard to provide additional security for such a system and only obfuscations methods may be increased (without using new hardware).

In [249] **Haiyang Sun et al.** describe that existing dynamic program analysis (DPA) options for Android that do not support generic tool creation and are mostly security focused. Also Android's multi-process architecture and missing APIs make it difficult for reengineers. In their research they developed a framework to support and simplify the development of generic DPA tools.

Comment: In fact, there are currently only a few tools available that can be used for dynamic analysis, including the Xposed framework, besides the usual tools such as gdb maybe. Their platform-independence and server/client approach are certainly beneficial.

In [250] Ashutosh Jain et al. illustrate methods for visualizing and detecting artifacts generated by obfuscation tools. They also noticed that files generated by apktool look completely different than the usual apps, since, e.g., its strings are not sorted anymore. In addition, their research discovered that only a few developers use obfuscation tools.

Comment: An interesting fact in terms of copy protection is the result they gained of the way apktool – often used for reengineering purposes – reorders the strings to an unsorted presentation. Furthermore, the figures that only 23 of 505 apps use obfuscation confirm our assumption about low security skills by many developers as described later.

9.5 Device- and user identification

Hristo Bojinov et al. [251] approached the hardware identification of mobile devices by taking a look at using various sensors for fingerprinting. In their paper they focused primarily on the acceleration sensor as well as the speaker and microphone. They approached the problem by measuring the tiny imperfections (noise) of these sensors that result from manufacturing processes. For instance, in a nutshell, and for the speaker/microphone case, they played sounds and recorded them directly to identify any patterns; meanwhile, they measured the Z-axis of the accelerometer to identity patterns for a certain device instead. They claim to have achieved an identification rate of 95% in the speaker/microphone case and up to 15.1% by using the acceleration sensor. Here, the weak rate may be improved by using additional information like the user-agent string to increase the identification rate up to 58.7%.

Comment:
While the identification of devices using sensors sounds interesting, it is quite difficult to apply it to the real world, and their research depends on further conditions. For instance, they

mentioned that the speaker/microphone case requires a quiet environment and depends on the actual surface on which the device is lying. Ultimately, it also requires two unusual Android permissions that make it less interesting for our copy protection. Instead, the identification rate for the acceleration sensor is quite weak, and they already noted for their demo application in the appendix that there is a significant interference of cables or objects under the phone that causes problems here.

In a similar research conducted by **Sanorita Dey et al.** [252] it was tried to identify smart devices by using the vibration motor in combination with the accelerometer. Here their results show that they were able to identify devices with an accuracy of up to 99% assuming sufficiently collected data.

Comment:
While a recognition rate of 99% sounds quite amazing, it has to be noted that this required 30 seconds of data collection. In terms of our topic of copyright protection it might drive a user crazy if the phone vibrates for such a long time. While their approach is certainly interesting, we need to summarize that it is not the identification option we are looking for.

In a research work by **Anupam Das et al.** [253] the speakers themselves were used for identification, while playing a clip on a smartphone and recording it at an external device. Under lab conditions, they were able to identify up to 94% of the clips/devices.

Comment: The solution is similar to the first research work of this section by Hristo Bojinov and limited to lab conditions, since they mention missing tests with different environments, the effect of the distance between source and recorder, and the presents of background noise.

Jan Lukas et al. [254] were one of the teams discovering the basics for identifying cameras based on a unique pattern noise. Their approach is based on the pixel nonuniformity noise that is caused by a different reaction of each pixel to light. They propose that this method is better than other methods using dark current noise maybe (cf. dark frames).

In **K. Kurosawa et al.** [255] the goal was to identify camcorders based on a unique pattern noise. They were able to obtain these pattern by examining about 100 frames where each device had its own pattern. Their method is based on dark current.

In [256], **Tomas Filler et al.** used the “photo-response non-uniformity (PRNU)” [[254] as quoted in [256]], “a multiplicative noise that is unintentionally embedded by the digital camera into every image” [256] , to identify camera models and brands by the fingerprints added by in-camera processing. Their method allowed correct identifications of up to 90.8% in their evaluation of 4500 cameras with 17 models and 8 different brands.

Comment: Using the camera sensor for identification purposes seems to be an interesting approach and Android offers access to existing images as well as the camera itself by obtaining the required permission. Depending on the method other factors such as no light during recording (dark frames) reduce the real world usage, of course. Instead other methods like PNU and PRNU sound promising.

9.6 Manipulation of sensors

By developing “SMASheD” [257] **Manar Mohamed et al.** created a framework to manipulate sensor data even on unrooted devices by using a native service executed via adb[78]. Besides the possible modification of sensor data, it can also be used to log various data, e.g., touch-sensor inputs. That way, it may be also used to control a device in any possible way that a user can do it. They also highlighted that it is possible to fake data provided by physical sensors that are often used for security purposes.

Comment: The fact that physical sensors may be overwritten on unrooted devices is alarming and affecting several security related applications that may use those sensors as their source for randomness. In terms of our user- and device identification section, the finally selected information sources (see evaluation) are not directly affected, but it highlights the suggestion of using multiple information source (including sensors) for additional security (see 11.4.1), while requiring all of them to be fulfilled or to have a low failure tolerance.

9.7 Section conclusion

Summarizing the available related work in the defined categories, it is noted that researchers are working on increasing the data/authentication security on Android by the introduction of SEs (e.g., Project Vault, etc.) and TEEs to Android devices as well as optimized Android versions. Most of that research is only beginning (cf. Google’s Trusty) and solutions are still being developed and researched, while available products sometimes show severe flaws (cf. exploits for QSEE / see 10.4.2).

Thinking about the more relevant topic of copy protection it is noticeable that, e.g., smartcards (similar to SEs) were already used in the past for licensing on desktop computers, while the encryption of software to prevent piracy has been known in the desktop world for years, in addition to the usage of native code. Researchers already adapted some solutions such as encryption and dynamic code loading for Android, but the presented ones are outdated by now due to the newly introduced ART VM around 2015. Those that are available, secure the actual transport of code (e.g., “Grab 'n Run” [223]), while they do not solve reengineering issues and related requirements (e.g., loading code from memory instead; see 11.4.3 for a proof-of-concept). In general, one can observe that solutions for desktop computers are much more advanced, and it is recommended to verify the possibilities of using these well-known options available to desktop computers on mobile operating systems as well. For instance, this most assuredly applies to obfuscation methods available to native code compilers outlined in 11.5.3.

In addition, the issues with Java code and its easy reengineering are not new either and obfuscation solutions have existed for years, while they were only adapted to Android in recent years. For instance, the encryption of strings belongs to these methods and one needs to be careful not to reinvent the wheel a second time.

[78] Android’s tool to connect to devices from a terminal via USB to perform various operations

There is also some research available on attacking Android itself and we need to highlight severe research results such as sensor value overwriting that affects many security solutions in theory (see 9.6).

In addition, the device and user identification have also been fairly well researched; however, the ideas need to be verified again for their usage on smartphone devices. For instance, the previously introduced PNU method was conducted by using professional camera equipment. It remains unknown if it is really usable on smartphones- and tablet devices even if it is assumed to work.

Last, but not least, Android's official copyright protections that are currently used on the major app markets (see 9.2.5) are not yet using secure solutions. Nevertheless, third parties are starting to focus on Android and offer solutions (see 9.2ff) using dongles, encryption, code loading or other options. Most of them seem to be presented at on some conferences in the past, but are not yet readily available to developers.

Therefore, we believe there is still some space for further improvements in terms of this work As explained in detail in the proposed solution sections (see 11.4ff), porting the existing LVL has not been done in advance, even other researchers invented their own licensing solutions. Moreover, our dedication to fuse Java and native code does not seem to be researched by others so far, even the idea of self-modifying code is not completely new. Moreover, third party researchers worked on general ideas like the whole encryption of apps instead. The same applies to the usage of smart cards that was already performed for mobiles and desktop systems, but in a slightly different way and by using other products.

10 Existing solutions and their challenges

This current section analyzes the most recent standing regarding copyright protection on Android by presenting general information and issues. Moreover, framework-specific problems based on the information provided in the fundamental section are extended here. In addition, ideas from the related work section are reviewed and some of those presented methods can no longer be used or have to be adapted. Furthermore, the possibility of using native code that is commonly known and possible on Android (by using its NDK) is also reviewed. Ultimately, evaluated examples of hardware protections such as SEs and TEEs are also found in this section.

10.1 Circumvention of default copyright protections on Android

This section covers an introduction to existing issues on current Android platforms related to copyright protection. It also gives an analysis on basic reengineering by gaining the program logic only (static analysis) and on the advanced reengineering used to obtain protocols besides the program logic (dynamic analysis). It covers the used tools and our approaches to circumvent current protections by the two major app markets – Google Play Store and Amazon’s App Store, in addition to presenting some of the major issues in general.

10.1.1 Copy protection means license verification

Nowadays, apps are mostly shipped digitally only, and customers download apps to their devices, while binding them to their accounts, e.g., the Google Play account. Therefore, modern copy protection describes more of the licensing of apps, and if a defined user is permitted (licensed) to use an app rather than actually owning it outright. This is probably Google’s reason for calling their protection library License Verification Library (LVL), while the previous version was initially a copy protection [258].

For that reason, a more generic issue regarding current Android platforms (like Google’s Play Store) is that APK files cannot be identified by their actual owner by viewing at the file only, because each APK file on any Android device is identically. Even if they are signed by the developer, it is not possible to distinguish between the APK files from different devices and copying them around is not prevented by Android. Moreover, it is not possible to implement any methods to verify if a user or device is allowed to execute the app and the question (by the implemented LVL in an app) is always if the logged on user is allowed to execute the app [47] after it was executed already.

So in summary, the raw copying of APK files is not prevented on Android and the general term ‘copy protection’ is a little misleading nowadays. It is a term that remains from several years ago when copy protection was really copy protection and applications were shipped on physical mediums.

In terms of our proposed methods, the missing identification possibility is an issue and addressed in 11.2.1 that shows options to apply kind of a real copy protection to apps again by binding the APK file itself to a user or device by adding these attributes to the file.

10.1.2 Android remains unsafe (rooting)

Rooting is a major issue for any type of protection and in particular, for hiding sensitive license data that need handling by an app, stored locally, or received upon runtime each time.

The reason to call it an issue is that it is a basic requirement (for reengineering) to access any app's private data or to intercept its communication with external servers; This would most likely be impossible without root rights. While the same issue exists on desktop computers, it is not considered a severe issue for them and all programs, especially protected ones, are only available in binary form (compiled native code), which is different than on Android.

Rooting an Android device is possible in two ways – either by permitted options through the manufacturer (e.g., Google's Nexus series allows it by default) or by using an exploit. So far, one can recognize at least one severe exploit that permits access to confidential data (or even root access) that is available every few weeks or few months; 2015 topped the lists with several severe exploits (see [25] [24]). Some of these exploits affected the kernel and even had the potential to breach sophisticated security measures like SEAndroid (aka[79] SELinux [52]).

Summarizing this information, hackers found vulnerabilities for nearly all Android versions and Android became infamously in the news headlines in 2015 for issues affecting billions of devices [259]. Just recently, another research team discovered several root exploits affecting millions of devices using Qualcomm chipsets once again [27].

The following diagram, by researchers from the University of Cambridge [260], shows an estimation on the number of vulnerable devices by major exploits. For doing so they selected 13 vulnerabilities (as of the writing of this text), including, e.g., "TowelRoot" and "Stagefright" and validated the available testing devices against these exploits by checking if the installed Android version is affected. The amount of testing devices was specified to be 21,713 that actually took "part in the Device Analyzer study" [260].

They categorized devices into three categories [260]:

"secure" – devices that "are not vulnerable to any of the vulnerabilities" [260]

"maybe secure" – devices with insecure Android versions that received a patch in general

"insecure" – devices with an insecure Android version and no available patches

Figure 48 illustrates their results that show the times with no secure devices at all, while – in general – and most often only the latest releases remained safe for a while. Of course, this is a severe issue assuming the commonly known fact that many carriers/manufacturers are still way behind the currently available version with their released phones.

[79] also known as

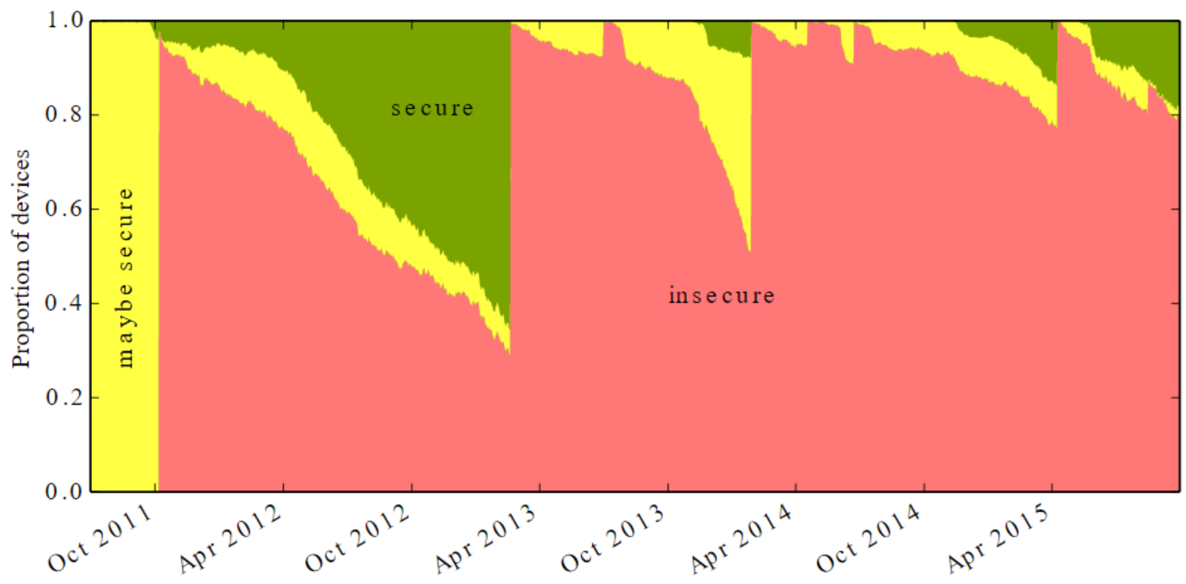


Figure 48 - Estimation on secure and insecure Android devices [260]

Conclusion on the rooting issue

From the current point of view, only the very latest Android versions are those that most often are secured against all major and newest root exploits. However, in watching the situation in the past, there are usually exploits right after the release of a new Android version (e.g., the Marshmallow-release in October 2015 [94] and “CVE-2015-6610”[80] shortly after). As outlined before, a privilege escalation is a severe threat in terms of copyright protection besides the permission/option by manufacturers to root a device and a customer may use the gained privileges to circumvent the protection.

In general, known as one of the benefits of Android, many technically skilled users prefer to root their smartphone to increase functionality. Depending on the required security level, companies need to decide if existing root access should be handled as a threat to an app and if its execution should be prevented. For instance, mobile banking apps sometimes have these requirements like the one by DKB, since it is simply more likely that the security is at risk on rooted devices (cf. reengineering, interception, malicious root apps like Trojans, etc.). In advance, it needs to be noted that the rooting may be hidden from apps by using specialized modules[81] available to the “Xposed Framework” [150]. It has to be concluded that a user can manipulate and intercept an app on a rooted device in almost any possible way using this tool. This is a huge issue for copy protection and an example is shown in 10.1.4.

10.1.3 Tools available to the general public

Besides the tools (see 8.2.2) that are available to developers, security engineers, or crackers, there are tools available for the usual public users, who often do not have the required skills to circumvent or crack any protections, but are enabled to do so by using the presented tools. This

[80] exploit - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6610
[81] http://repo.xposed.info/module/com.devadvance.rootcloak2

section does not only introduce an example for one of these tools, but it also tries to present the details so that developers can take precautions against it.

Lucky Patcher

Lucky Patcher[82] (LP) is a tool that may be categorized in the basic reengineering section that is available to the general public and to all those, who are looking for tools to circumvent Android's license protection and perhaps others. It is able to patch almost any app protected by Google's LVL, Amazon's DRM, or Samsung's solutions based on universal patches and custom recipes. It also includes general cracking solutions that modify the underlying system. While it includes a feature to crack APK files for redistribution, it usually works on the optimized version of an app stored in the /data/dalvik-cache directory for restoring purposes. This makes it harder for developers to detect any manipulations by perhaps using checksums. Here, the reason is that the optimized version depends on the device's attributes and is always different [64, p. 62ff] [66, p. 30ff].

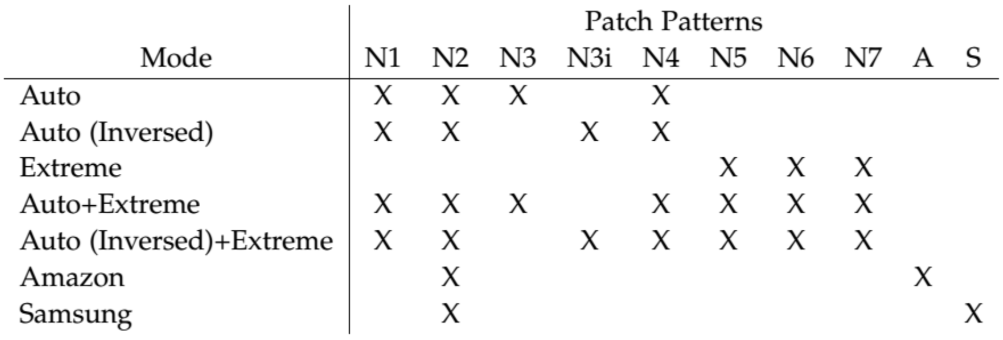
As described in [66, p. 33ff], a blackbox approach was used to analyzed LP due to the unavailable source codes and protections that made it time-consuming to totally reengineer it. The blackbox approach provided a sufficient outcome to gain an insight on LP's efforts to crack the protection schemes. Also, the application provides itself some relevant information on the used patches.

As outlined in [66, p. 34] LP provides several modes to crack applications:

- The "Auto Mode" using a minimal amount of patches to circumvent apps with basic protection
- The "Auto Mode (Inversed)" with a similar functionality as the one before and with slight changes
- The "Other Patches (Extreme Mode!)" using further available patches online
- The "Auto Mode (Amazon Market)" for disabling Amazon App's DRM protection
- The "Auto Mode (SamsungApps)" for disabling the protection of Apps from Galaxy Apps[83]

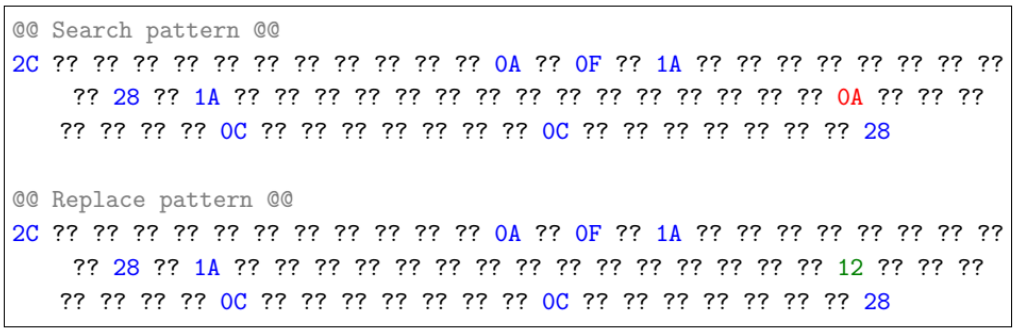
According to [66, p. 35ff] LP provides seven modes that may apply up to 10 so-called 'patch patterns' (described below) as shown in Figure 49; each patch pattern may come with a series of so called 'search patterns' as illustrated in an example in Figure 50. The patch patterns N1 to N7 target Google's LVL, while the patch pattern A relates to Amazon's DRM, and S to Samsung Apps. The patching takes place on the bytecode level and according to the chosen mode (see above), LP tries to apply the patches using the search patterns to modify a series of bytes to have the desired result (e.g., valid license or an ignored license code, etc.).

[82] http://lucky-patcher.netbew.com/
[83] Samsung's App Store



Mode	Patch Patterns									
	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

Figure 49 - LP Modes and related Patch Patterns [66, p. 35]



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

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

Figure 50 - Search and Replace Patterns with blue fixpoints and placeholders marked as ?? [84] [66, p. 36]

The following table shows the high-level details of each patch pattern taken from [66, p. 37ff] in a summarized way, while further details, e.g., on actual bytecode changes, may be obtained from the original work, if necessary. A basic knowledge about the LVL as found in the fundamental section (see 8.3.2), as well as further information from the licensing reference by Google (see [176]), are required to understand the following patch pattern descriptions in full detail, even short descriptions on the responsibilities of target classes are provided inline.

Patch Pattern	**Modification**	**Responsibilities / Result**
N1	switch statements in verify() method of LicenseValiditor class modified	Responsibilities: Method decrypts and verifies response from license server [176] Result: Case "LICENSED" and "NOT_LICENSED" are treated as valid now. Instead "LICENSE_OLD_KEY" that should be fine (updated signature by developer), triggers an error now.
N2	if-statement disabled in verify() method of LicenseValiditor class modified	Responsibilities: Method decrypts and verifies response from license server [176]

[84] Mr. Neutze chose that hexadecimal value representation for improved readability, e.g. 0f instead of 0x0f

		Result: The result of the signature verification is ignored and the program flow continues as if the signature was valid
N3	return-value of method allowAccess() set to “true” inside APKExpansionPolicy and ServerManagedPolicy class modified	Responsibilities: Default policies that e.g. manage storage of license data/validity etc. [176] Result: The app may be cracked, but the user has to verify the success of the operation
N3i	return-value of method allowAccess() set to “false” inside APKExpansionPolicy and ServerManagedPolicy class modified	Responsibilities: Default policies that e.g. manage storage of license data/validity etc. [176] Result: The app may be cracked, but the user has to verify the success of the operation
N4	if-statement in LicenseChecker class modified that initiates license check of checkAccess() method by modifying the condition to an inequality check to two calls of mPolicy.allow(), which is always false now	Responsibilities: Class used for initiation of license check [176] Result: mPolicy.allow() result ignored and due to inequality check the execution of the condition block prevented. It means the result of checkAccess() method is not considered.
N5	if-statement in verify() method of LicenseValidator class modified by changing condition to a comparison with LICENSED (value = 0)	Responsibilities: Method decrypts and verifies response from license server [176] Result: The response code by the server is still parsed, but its result ignored
N6	if-statement and variable in verify() method of LicenseValidator class modified by setting responseCode to LICENSED and modification of if-statement so that it can never execute	Responsibilities: Method decrypts and verifies response from license server [176] Result: This prevents the verify() method from handling cases that are neither LICENSE_OLD_KEY, NOT_LICENSED nor LICENSED.
N7	variable exchange in verifyLicense() in onTransact() of	Responsibilities: ILicenseResultListener (IPC

	ILicenseResultListener class as well as all classes of /com/android/ package. This is done by exchanging the responseCode with LICENSED (value = 0)	Callback) handles asynchronous replies from the license server [176] Result: License code manipulated. Furthermore the patch pattern (=initializing a variable instead of moving a result) may apply elsewhere, which may lead to instability.
A	if-statement in obfuscated class com/amazon/android/licensing/b.java modified by inequality check of a comparison of the same string as well as modifications of comparisons for checking strings for not being null in com/amazon/android/o/d.java	Responsibilities: b.java verifies the license and d.java the license expiration Result: b.java / Condition, if application is licensed, is always licensed now. d.java / obfuscated function returns true always
S	if-statement and return value modification in obfuscated LicenseRetriever class and Zirkonia class' checkerThreadWorker() method by executing LicenseRetriever's receiveResponse() method, but comparing v0 to itself and modifying the result variable in Zirkonia class to true instead of using the actual receiveResponse()'s return value	Responsibilities: License checks and local storage of license Result: License file checks deactivated by returning true within the verification always. Also any other response code than LICENSED is accepted.

Table 3- Patch Patterns, high level modifications and its results (based on [66, p. 37ff])

10.1.4 Static analysis and disassembler tools (Dalvik bytecode)

As presented in the fundamental section (see 8.2.1) already, it is fairly easy to reengineer Android applications up to the most recent Android versions with some training on smali.

The only requirement to do this is the access to the DEX file (classes.dex) that is usually available within the APK files to reveal its internal program logic using the aforementioned tools (see 8.2.2), and to transform the bytecode to assembly (smali) or even Java code. Depending on the used obfuscation tools by an app developer and actual obfuscation level, the only limiting factor is time.

In this section, we want to review Amazon's DRM that can already be circumvented by basic reengineering techniques. In a requested thesis in order to analyze used protections on Android it was discovered [64, p. 30ff] that all applications (of the Amazon AppStore) are installed into the less-protected directory /data/app and may be easily received on any device. Using the aforementioned reengineering tools, the protected apps reveal themselves with several additional framework codes by Amazon, in addition to the actual program logic. Even given the fact that the code is protected by obfuscation tools (as seen in Figure 51), it is noticeable that not all code is obfuscated, especially the “com.amazon.android.Kiwi” [64, p. 31] class and related files. By default, the AndroidManifest.xml file reveals the initially launched activity by Android for an app. It appears that Amazon adds calls to the “com.amazon.android.Kiwi” in the “onCreate, onPause, onResume, onStop, [and] onCreateDialog” [64, p. 31] methods of that file, which redirect to a “preprocess” [64, p. 31] method that is not involved in the app logic itself. We assume that Amazon clearly separates the app logic from its DRM code, which makes it much easier for attackers to separate it.

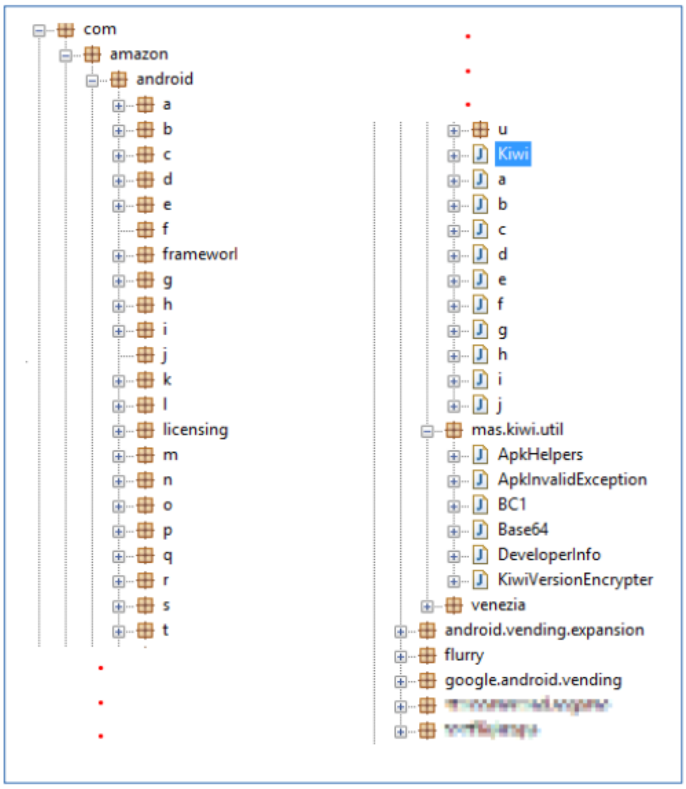


Figure 51 - Obfuscated code by Amazon added to an app (extract) [64, p. 31]

The actual cracking process [64, p. 32] was done (in 2014) by simply removing all calls in the launching activity towards the aforementioned kiwi class, as well as any references to that file, in several files of the “com.amazon.android” package. Here, the “calls are routed indirectly to the static method addCommandToCommandTaskPipeline from the Kiwi class” [64, p. 32]. It activates Amazon’s key verification in another file. For deactivating the verification and ultimately cracking the protection, the calls to the “addCommandToCommandTaskPipeline” [64, p. 32] method simply needed removal.

While Amazon did not change its way of protection for a long time (as reviewed and confirmed in another requested student’s thesis in 2015 [75, p. 33]), recently renewed analyses by the author himself (March 2016) revealed that Amazon modified it and the aforementioned approach did not seem to work anymore. Nevertheless, basic investigations allowed us to assume that the new protection mechanism offers different flaws. For instance, even a failed-license message appeared after uninstallation of the related App Store app, but the tested app itself continued to run and was even controllable. Approaches that just tried to hide the message, triggered a CRC check and error, but in theory, it could have been sufficient to hide that message.

Therefore, students were asked to conduct further investigations [241] while participating in our research in the Android Practical Course. It was then discovered that besides the removal of the aforementioned invocations of “addCommandToCommandTaskPipeline” [241] within the KiWi class, the constructor had to be modified to set a Boolean variable DRMenabled to false in order to deactivate the protection, which was once more verified by the author with another app in June 2016. The whole cracking is essentially a task of several seconds for an attacker only. Also, Amazon was notified about this issue and its upcoming release with this dissertation (see 14.5). We assume that it is now fixed.

10.1.5 Dynamic analysis and tools for interception/manipulation (Dalvik bytecode)

Besides the decompilation of Android Apps, a more sophisticated attack is to analyze the traffic on network- and local interfaces as well as between apps and services by an app (cf. function calls). Since Google infamously made it to the headlines in 2014 for missing advisable MITM protection in its Gmail app [[261] as quoted in [262]], the author of this dissertation requested in a related student thesis (cf. [64]) to look for similar flaws. Ultimately, the tool Xposed framework[85] was discovered – and by that time, it was mostly used for tiny modifications of apps as well as by the aforementioned student’s thesis in order to analyze the communication of the LVL with other services and ultimately to circumvent it.

Technically, the Xposed Framework, as outlined in [131], replaces a system file on a rooted device namely, the “/system/bin/app_process” [131] with an extended version, and therefore allows it to “act in the context of the Zygote process” [64, p. 54] that runs with root privileges. Meanwhile, all applications are its child processes as shown earlier in the fundamental section in Figure 16 (see 7.3.7). Therefore, all newly spawned child processes inherit the Xposed capabilities and can be modified dynamically now.

Developers, or here aka the ‘attackers’, can create modules for this framework that are configured within the framework’s management tool. “The power of XPosed [sic!] comes from the fact that it is able to intercept any Java method call in the context of the current Zygote process.” [64, p. 54]. For instance, the framework allows the following: the manipulation of any app, changes to its layout, and changes to its behavior, and all of it “on-the-fly”. It is used to extend app functionalities, remove limitations or (as in the following case) disable the license verification.

The LVL by Google is available as an example code for integration[86] into anyone’s own project as previously introduced in 8.3.2. Therefore, all required methods to disable, manipulate the encrypted, and in theory, protected license responses, are publicly known. The novel approach of the following method is that it works on-the-fly, and in theory with any app implementing the LVL in the default way. Unlicensed apps will receive a valid signature (faked, valid license response) and there is not any way for an app or Google to detect if this actual attack has taken place [64, p. 54ff]. Of course, it is possible to implement special countermeasures against this hooking assuming that such an attack could take place and assuming app developer know about it. These ideas are presented in 11.5.2 as part of our proposals.

[85] http://repo.xposed.info/
[86] Instruction: http://developer.android.com/google/play/licensing/setting-up.html

As outlined earlier, we discovered this issue while investigating possible MITM attacks to underline our theory that the current protection library by Google needs further improvement. For attacking the actual implementation, it is required to first review the license requests in a more detailed way (see 8.3.2 for fundamental details first).

By default, the app formulates a license request, which includes a "timestamp and a nonce. These values must also be present in the response to guarantee […] no man-in-the-middle attack[s]" [64, p. 55]. When a developer "registers a new app to use the LVL […] a new pair of public/private keys is generated. The developer only gets to see the public key [cf. developer console], which must be embedded in the application" [64, p. 55].

Any license response's signature receives validation with that public key, which only returns a match, when it is signed by the private key. Only Google has access to the private key. In the event that this is successful the actual response is parsed [64, p. 55]. The proper license check and interpretation takes place "in the ILicensingService interface […] [, where] the Stub class extends the android.os.Binder class and […] must override the onTransact method. This is the entry point of the licensing response that comes from the [Google] PLAY STORE [sic!] app. The response data are packaged as an android.os.Parcel containing […] [an] integer representing the server response code" [64, p. 55] (e.g., 0x00 for licensed / cf. [87]). This whole LVL request using the binder is illustrated in Figure 52 showing the initial registration from the LVL service (1 to 7), to the actual license request (step 8ff) of an app that consists of marshalled

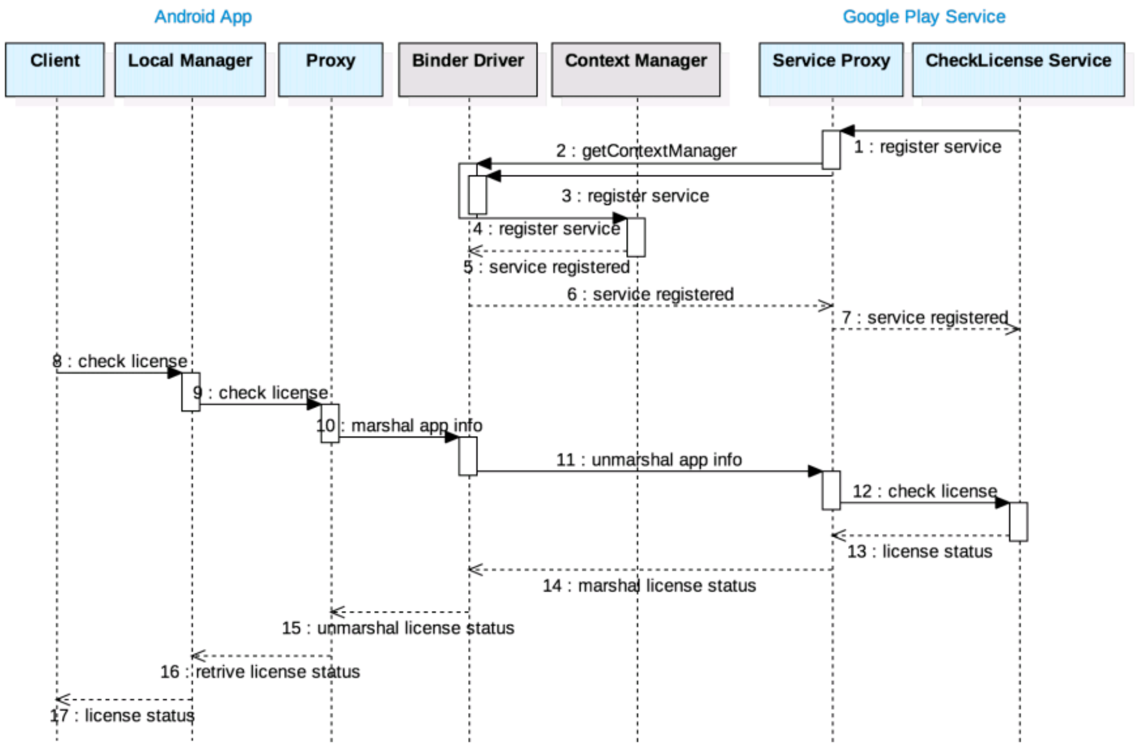


Figure 52- Communication of a performed license check using the LVL [62, p. 22] (based on [263])

information (see 11.4.8 for all reengineered details) that get transported to Google's license server to reply with the license status finally [62, p. 22].

[87] https://code.google.com/p/marketlicensing/source/browse/library/src/com/android/vending/licensing/LicenseValidator.java

Returning to the aforementioned parcel, it contains “A string representing the server response data, composed of six concatenated values using the symbol ‘|’ as a delimiter” [64, p. 55]. These six values are the actual response code, the nonce, the package name, the “version code of the app”, “an app-specific user id” and the “timestamp included in the request” [64, p. 55].

Additionally, there is a “Base64-encoded string representing the signature of the previous string of concatenated values (for authenticity)” [64, p. 56].

In the mentioned “onTransact method the data is extracted from the Parcel and forwarded to an instance of ”the LicenseValidator for checking, interpreting and actual handling” [64, p. 56], which takes places “in the verify (…) method” [64, p. 56]. A valid license requires 0x00 as a response code [64, p. 56].

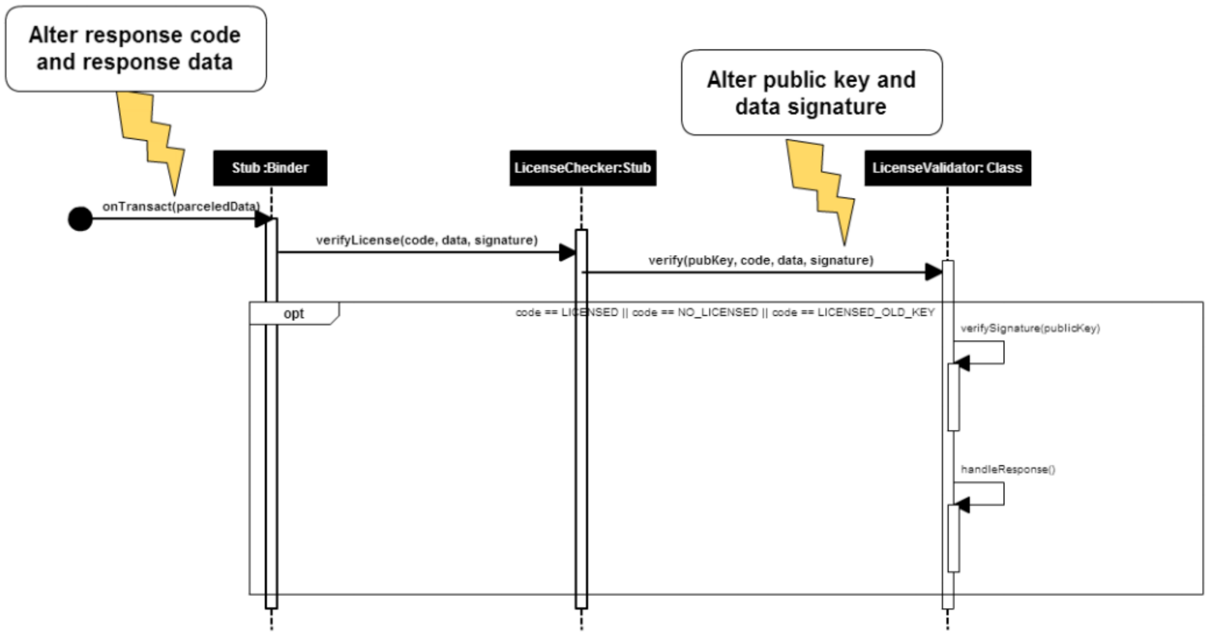
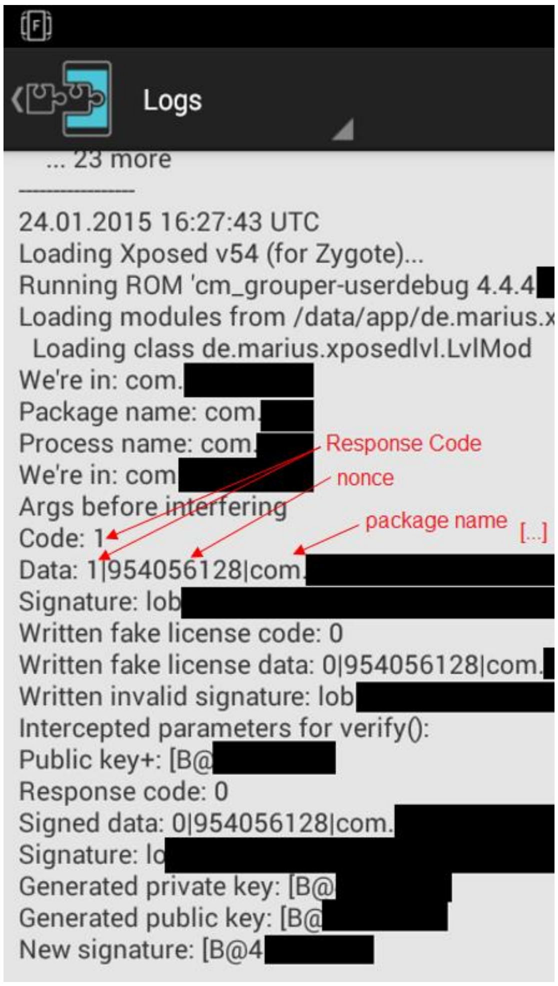

Alter response code and response data
Alter public key and data signature
Stub :Binder
LicenseChecker:Stub
LicenseValidator: Class
onTransact(parceledData)
verifyLicense(code, data, signature)
verify(pubKey, code, data, signature)
opt
code == LICENSED || code == NO_LICENSED || code == LICENSED_OLD_KEY
verifySignature(publicKey)
handleResponse()

Figure 53 - Handling of the server response by Google and manipulated methods [64, p. 57]

For attacking the LVL, “The OnTransact method [see Figure 53] is the first one that needs to be intercepted. Its second parameter is the Parceled [sic!] data, containing the server response code, the response data, and its signature. This is the parameter that needs to be altered” [64, p. 57] by creating a new parcel.

It is “created and the string ‘com.android.vending.licensing.ILicenseResultListener’ is written to it as an interface token. This assures the receiver that the Parcel [sic!] is really intended for it. Next, the value 0 is written to it […] (= licensed) […] The next value is the response string, which is copied directly from the old Parcel [sic!], except that the first concatenated value is overwritten with a 0. The final value in the new Parcel [sic!] is the signature of the previous string” [64, p. 57]. Due to the missing private key, it cannot be computed here. In the end, “the data position in the new Parcel [sic!] is reset and the new parameters are forwarded to the onTransact method” [64, p. 57].

Since the modified response’s signature is not valid yet, another method needs to be intercepted. The signatures are validated within the LicenseValidator’s verify() method. For “passing this test, the public key and the signature of the new server data [(=faked data)], which the method receives as parameters, must match” [64, p. 58]. This can be done by “generating a new pair of public/private keys […] [where] the private key” is used to compute the valid signature for the faked response. “Finally the newly generated public key and the newly computed signature are forwarded to the verify (…) method” [64, p. 59]. Figure 54 shows the logging output within the Xposed Framework tool that represents the above in a visualized way hiding the cracked app name and other details.



```
... 23 more
-----------------
24.01.2015 16:27:43 UTC
Loading Xposed v54 (for Zygote)...
Running ROM 'cm_grouper-userdebug 4.4.4[redacted]
Loading modules from /data/app/de.marius.x
  Loading class de.marius.xposedlvl.LvlMod
We're in: com.[redacted]
Package name: com.[redacted]
Process name: com.[redacted]
We're in: com[redacted]
Args before interfering
Code: 1
Data: 1|954056128|com.[redacted]
Signature: lob[redacted]
Written fake license code: 0
Written fake license data: 0|954056128|com.[redacted]
Written invalid signature: lob[redacted]
Intercepted parameters for verify():
Public key+: [B@[redacted]
Response code: 0
Signed data: 0|954056128|com.[redacted]
Signature: lo[redacted]
Generated private key: [B@[redacted]
Generated public key: [B@[redacted]
New signature: [B@4[redacted]
```

Figure 54 – Xposed’s logging of an attack by the developed Xposed Module for LVL circumvention (app information hidden)

While Google’s servers are aware of this license request (and replied with a license code representing an invalid license), the LVL as well as the app will receive a legitimate reply with the status code “licensed”, which equals 0x00. The code used for the above attack may be found in the Appendix (see 15.1.1). There is no way for an unprepared app to know that this actual attack took place yet [64, p. 59] and the app execution is simply paused by Xposed and continued afterwards. 11.5.2 illustrates some options to recognize possible Xposed attacks.

In addition, our research to improve the LVL by porting it to a native code version called “nLVL” [62] was undertaken using, e.g., the above Xposed tools for analyzing and reengineering the secured communication between the Google Play Services and the license servers by Google, as explained by [62] in its LVL analysis section. That this was even possible, reveals how seriously insecure Java-only implementations are and the must requirement for alternative solutions.

10.1.6 Further options of Xposed Framework

The previously shown attack on the LVL shows the possibilities of the Xposed framework that was raised in recent years to the default tool used by modders to apply all kinds of methods to existing apps from simple UI changes to the removing of limitations. A large community provides more than 887 Xposed modules by now (June 2016) [264].

Particularly of interest to this work, besides our own module, are modules affecting the DRM of Android. For instance, a module [265] by the author "veetip" disables the new secure flag used by many applications to block the taking of screenshots by a user. It needs to once again firmly stated that anything executed on Android is not secure; it can be intercepted and modified – no matter how good an encryption might be. While Java implementation are extremely easy to reengineer, native code implementations provide attackers much less information. The Android NDK is analyzed separately (see 10.3).

10.1.7 Section conclusion

As outlined previously, root access on Android is quite common. Moreover, the available copyright protection mechanisms used by the major app markets are seriously broken. While Amazon supports an interesting way of integrating the protection automatically, Google's Play Store developers have to implement everything manually and it depends on the skills of these developers to increase or decrease the difficulty in terms of cracking an app. This also applies to Samsung's protection library Zirconia as well as SlideMe's SlideLock presented in the related work section, since their implementations are very similar.

Solutions using native code, as introduced in the related work sections, were not considered and their usage by developers remains unknown. It is assumed that only a few developers are aware of the issues in general and even fewer may apply sophisticated protections. Eric Lafortune, CTO of GuardSquare, and developer of Pro- and DexGuard, shares this assumption. He also provided us "some statistics on the protection of the top European banking apps. It seems that about 65% use ProGuard, 15% use DexGuard, and 20% are unprotected […] Considering that ProGuard only offers very basic protection (name obfuscation), most developers indeed seem to be unaware [of the problems related to Android reengineering]" [35]. In addition, Ashutosh Jain et al. [250] confirm with their research results based in 2015 that only a few developers used obfuscation tools. Only 23 out of 505 apps (less than 5%) were protected by DexGuard or ProGuard, while, even worse, 400 of these apps included debug information[88]. At this point, it needs to be noted that they used the top free apps from each category and developers here (instead to commercial companies) may not try to gain profit and protect their apps (exception: freemiums[89]).

[88] This allows to restore the source code quite well
[89] Apps that appear to be free, but offer paid services

10.2 Available copyright protections by third parties for Android

The following section should review the solutions presented by other researchers and companies or general solutions that are available today and intended to prevent or identify illegal copies of an app. These were already presented in detail, as well as with an initial comments, in the related work section (see 9ff). Research work requiring system modifications is out of the scope of this dissertation and not reviewed, since a major goal of this dissertation is to highlight solutions for existing platforms.

10.2.1 Solutions for dynamic code loading

Many of the presented solutions that use dynamic code loading are no longer usable on newer Android versions or with reduced security protection in addition to requiring modifications (see 11.4.3 for details and possible options). This applies to the VM-based idea by Wu Zhou et al. [162], the encryption/server-based approach by Sung Ryul Kim et al. [236] as well as the encryption/license solution based on smartcards by Shoaib et al. [229]. Here the reason is because Google introduced the ART VM that requires valid opcodes besides removing the DexLoader function accepting a byte array to load code from memory [100, p. 31]. Nowadays, it is required to store the code in a file on a disk first, in order to load it upon runtime. This decreases the protection extremely, and it would have been much harder to extract the DEX file from memory instead of watching for any DEX files (and their optimized code versions created by Android) in the private directory of an app.

10.2.2 Solution for identifications

Reviewing methods that merely identify manipulations by fingerprinting or watermarking using different methods as those presented by Joohyouk Jang et al. [234] and Hyunho Ju et al. [235] and Hugo Gonzales et al. [225] may still be used, but do not represent full copy protection mechanism like the license protection solutions introduced earlier.

10.2.3 Solutions to prevent reengineering

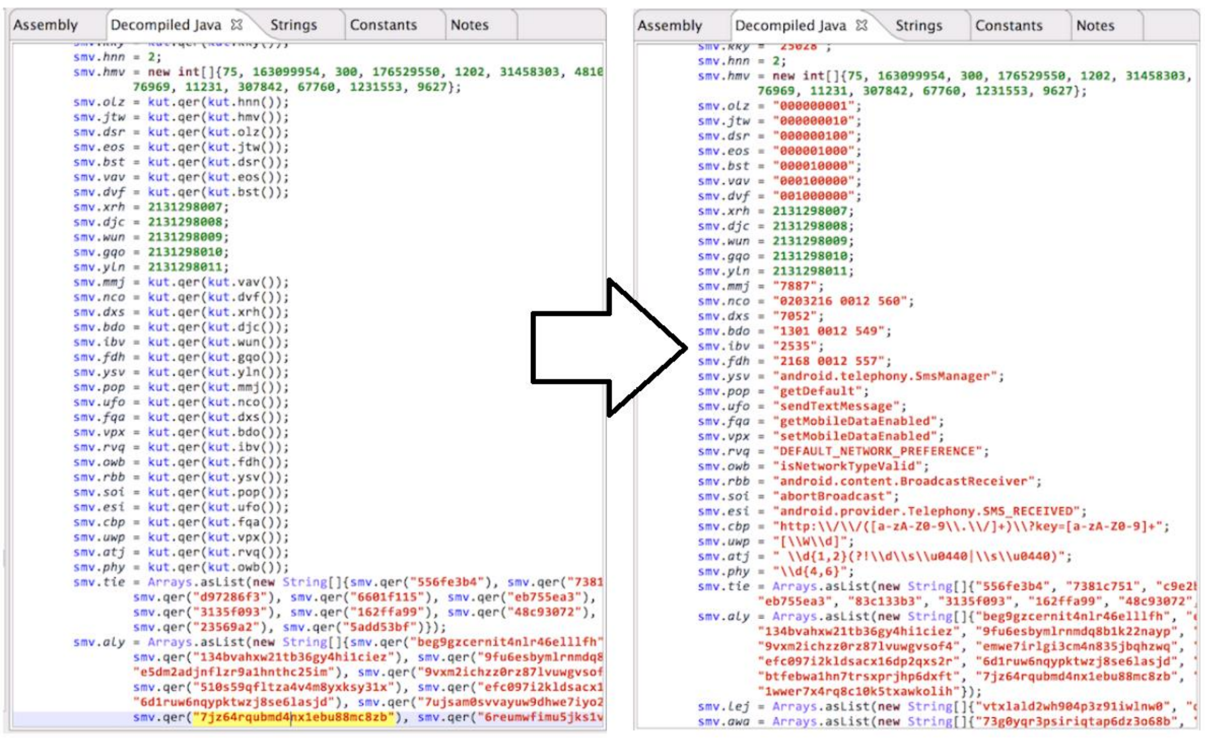
No full copy protection also applies to methods preventing (=increasing time) analyses as presented by Thansis Petsas et al. [224] as well as for self-modifying code ideas by Daniel Hugenroth et al. [226] and ultimately, the obfuscation solutions like ProGuard [161] and DexGuard [160], too.

Additionally, well-known obfuscation techniques like JunkBytes [266] (based on [267]) that make use of faulty opcodes that were not executed by Android, but which tried to be interpreted by disassemblers that cause them to stop working, are not usable on modern Android systems anymore due to the pre-compilation requirement for ART VM [100] as well as the fixed bugs according to [233]. Also, recent verifications using the supplied app from [268], show that it worked on Android versions prior to ART VM, while crashing the app on modern Android versions as expected by the author.

Furthermore, the ‘hidden method’ invocation, as introduced in [64, p. 82] (based on [269]) cannot be used securely on modern Android versions using ART VM, since the OpenDexFile(Byte[] …) function to open DEX code from memory got removed [100, p. 31] and the remaining openDexFile(File…) function requires a local file that would tremendously decrease the security benefit. Due to ART’s pre-compilation, ‘hidden methods’ are already ignored for the compilation now and certainly will be ignored in the future. They are probably not copied to OAT file’s embedded DEX structure by today (not verified; solution not of interest due to the mentioned issue above).

Moreover, it is important to note that current obfuscation techniques like ProGuard are the target of deobfuscation tools like “Simplify” [270] and “Oracle” [271] that try to bring some logic back to the obfuscated code (see Figure 55 for an example). As of now, these approaches provide limited results, and the obfuscation still needs to be analyzed by a human instead.

While some of the mentioned approaches may still be used, but do not represent a full copy protection solution yet, we aim to include the one or another to offer a full solution as outlined in the proposed solution section (see 11.4).



```
        smv.hnn = 2;
        smv.hmv = new int[]{75, 163099954, 300, 176529550, 1202, 31458303, 481e
                76969, 11231, 307842, 67760, 1231553, 9627};
        smv.olz = kut.qer(kut.hnn());
        smv.jtw = kut.qer(kut.hmv());
        smv.dsr = kut.qer(kut.olz());
        smv.eos = kut.qer(kut.jtw());
        smv.bst = kut.qer(kut.dsr());
        smv.vav = kut.qer(kut.eos());
        smv.dvf = kut.qer(kut.bst());
        smv.xrh = 2131298007;
        smv.djc = 2131298008;
        smv.wun = 2131298009;
        smv.gqo = 2131298010;
        smv.yln = 2131298011;
        smv.mmj = kut.qer(kut.vav());
        smv.nco = kut.qer(kut.dvf());
        smv.dxs = kut.qer(kut.xrh());
        smv.bdo = kut.qer(kut.djc());
        smv.ibv = kut.qer(kut.wun());
        smv.fdh = kut.qer(kut.gqo());
        smv.ysv = kut.qer(kut.yln());
        smv.pop = kut.qer(kut.mmj());
        smv.ufo = kut.qer(kut.nco());
        smv.fqa = kut.qer(kut.dxs());
        smv.vpx = kut.qer(kut.bdo());
        smv.rvq = kut.qer(kut.ibv());
        smv.owb = kut.qer(kut.fdh());
        smv.rbb = kut.qer(kut.ysv());
        smv.soi = kut.qer(kut.pop());
        smv.esi = kut.qer(kut.ufo());
        smv.cbp = kut.qer(kut.fqa());
        smv.uwp = kut.qer(kut.vpx());
        smv.atj = kut.qer(kut.rvq());
        smv.phy = kut.qer(kut.owb());
        smv.tie = Arrays.asList(new String[]{smv.qer("556fe3b4"), smv.qer("7381
                smv.qer("d97286f3"), smv.qer("6601f115"), smv.qer("eb755ea3"),
                smv.qer("3135f093"), smv.qer("162ffa99"), smv.qer("48c93072"),
                smv.qer("23569a2"), smv.qer("5add53bf")});
        smv.aly = Arrays.asList(new String[]{smv.qer("beg9gzcernit4nlr46elllfh"
                smv.qer("134bvahxw21tb36gy4hi1ciez"), smv.qer("9fu6esbymlrnmdq8
                "e5dm2adjnflzr9a1hnthc25im"), smv.qer("9vxm2ichzz0rz87lvuwgvsof
                smv.qer("510s59qfltza4v4m8yxksy31x"), smv.qer("efc097i2kldsacx1
                "6d1ruw6nqypktwzj8se6lasjd"), smv.qer("7ujsam0svvayuw9dhwe7iyo2
                smv.qer("7jz64rqubmd4nx1ebu88mc8zb"), smv.qer("6reumwfimu5jks1v
```

```
        smv.hnn = 2;
        smv.hmv = new int[]{75, 163099954, 300, 176529550, 1202, 31458303,
                76969, 11231, 307842, 67760, 1231553, 9627};
        smv.olz = "000000001";
        smv.jtw = "000000010";
        smv.dsr = "000000100";
        smv.eos = "000001000";
        smv.bst = "000010000";
        smv.vav = "000100000";
        smv.dvf = "001000000";
        smv.xrh = 2131298007;
        smv.djc = 2131298008;
        smv.wun = 2131298009;
        smv.gqo = 2131298010;
        smv.yln = 2131298011;
        smv.mmj = "7887";
        smv.nco = "0203216 0012 560";
        smv.dxs = "7052";
        smv.bdo = "1301 0012 549";
        smv.ibv = "2535";
        smv.fdh = "2168 0012 557";
        smv.ysv = "android.telephony.SmsManager";
        smv.pop = "getDefault";
        smv.ufo = "sendTextMessage";
        smv.fqa = "getMobileDataEnabled";
        smv.vpx = "setMobileDataEnabled";
        smv.rvq = "DEFAULT_NETWORK_PREFERENCE";
        smv.owb = "isNetworkTypeValid";
        smv.rbb = "android.content.BroadcastReceiver";
        smv.soi = "abortBroadcast";
        smv.esi = "android.provider.Telephony.SMS_RECEIVED";
        smv.cbp = "http:\\/\\/([a-zA-Z0-9\\.\\/]+)\\?key=[a-zA-Z0-9]+";
        smv.uwp = "[\\W\\d]";
        smv.atj = " \\d{1,2}(?!\\d\\s\\u0440|\\s\\u0440)";
        smv.phy = "\\d{4,6}";
        smv.tie = Arrays.asList(new String[]{"556fe3b4", "7381c751", "c9e2
                "eb755ea3", "83c133b3", "3135f093", "162ffa99", "48c93072",
        smv.aly = Arrays.asList(new String[]{"beg9gzcernit4nlr46elllfh", "
                "134bvahxw21tb36gy4hi1ciez", "9fu6esbymlrnmdq8b1k22nayp",
                "9vxm2ichzz0rz87lvuwgvsof4", "emwe7irlgi3cm4n835jbqhzwq",
                "efc097i2kldsacx16dp2qxs2r", "6d1ruw6nqypktwzj8se6lasjd",
                "btfebwa1hn7trsxprjhp6dxft", "7jz64rqubmd4nx1ebu88mc8zb",
                "1wwer7x4rq8c10k5txawkolih"});
        smv.lej = Arrays.asList(new String[]{"vtxlald2wh904p3z91iwlnw0", "
        smv.awa = Arrays.asList(new String[]{"73g0yqr3psiriqtap6dz3o68b",
```

Figure 55 - Example for Simplify conversion (based on graphics from [270])

10.2.4 Existing copyright protection and DRM solutions

In addition to the more deeply analyzed solutions by Google (LVL) and Amazon (DRM) earlier, other app markets provide different solutions. For instance, “Zirkonia” by Samsung is a full copyright protection solution, but faces the same issues as Google’s LVL solution since it was implemented almost in the same manner, and manually crackable by Lucky Patcher as analyzed in [66, p. 22f].

Instead, multimedia distributors profit from sophisticated DRM solutions by Google called “Widevine” [33] for their media content released on various video platforms and even integrated TAs in Trusted Execution Environments by major distributors [205]. Usually, app developers have no access to this protection level. Either they need to apply for it themselves and choose the target TEE (license costs!) or have to develop their very own solution (e.g, our proposal of using SEs).

10.2.5 Native code copyright solutions

Since Google does not recommend [7] to use the NDK, there are no available native copyright protection mechanism by Google available. Also, Amazon or other markets seem not to provide any native code solutions yet.

Nevertheless, as outlined in the related work section (see 9.2ff), researchers already created several solutions for copyright protection using native code for obfuscation reasons in their solutions. These solutions were examined previously (see 10.2.1), while the general benefit of using native code is analyzed in its own subsection next (see 10.3).

10.2.6 Section conclusion

There are few available solutions that work with modern Android versions, and most of the existing solutions either require an update (cf. removed dynamic code loading functionality) or are seriously broken (e.g. Google’s LVL, Amazon’s DRM, etc.).

Nevertheless combinations of existing solutions and the implementation of provided license verifications in a recommended way (cf. best practices and Google’s request of modifying the LVL implementation [47]) in combination with tools like ProGuard [161]or better DexGuard [272] can increase the security protection already. A remaining issue is that all the framework calls remain visible and reveal the functionality of an obfuscated file most often, even it requires more time to understand complex codes in all details.

Ultimately, the main issue remains that DEX files can be reengineered easily and Google shows no intention on removing that threat anytime soon, since DEX files are even embedded in OAT files used by the newer ART VM again [100, p. 18].

10.3 Using native code (Android NDK/ARM binaries)

The idea of using native code is not new and Google permitted developers the inclusion of native code parts using the Android NDK since the early days of Android. The general question, whether or not to use it for obfuscation already came up earlier (e.g., [273]) in addition to the ideas that are based on using native code for protection (e.g., [162]).

By default, it needs to be noted that native ARM binaries do not include as many references as the Dalvik Bytecode anymore. Moreover, it needs to be understood that the decompilation of binaries is much harder, and (as introduced in [274] about the requirements for decompilation) that, e.g., "information about the originally used programming language and compiler is valuable during the decompilation process because each compiler generates quite unique code" [274]. Since many references are missing, a decompiler is required to recover, e.g., "local variables, used ABI[90], functions and their arguments" [274], as well as to perform a "reconstruction of high-level control-flow constructs, such as loops and conditional statements". Furthermore, "a sequence of machine-code instructions" [274] needs to be translated back to a high-level instruction. Nevertheless, the authors of that work and developer of the Retro Decompiler claimed to achieve "comparable [results] with […] existing commercial non-targetable decompilers, such as Hex-Rays decompiler [and] […] achieve over 90% accuracy of successfully recovered functions and 91% of recovered function arguments" [274].

Therefore, their decompilation tool[91] is used in the following chapters to make assumptions on how easy or difficult it is to reengineer typical native code, while the ARM assembly code itself is considered secure enough (cf. majority of CS students not even familiar with it, see 10.3.5).

10.3.1 Simple native code example

The conducted experiments using simple C source code examples as shown in Table 45 and Table 43 in the Appendix (see 15.1.3 and 15.1.4) reveal the issues with decompilation of ARM binaries using maximum optimizations during its compilation. Viewing their corresponding decompilation results in Table 46 and Table 44, it is certainly already beneficial to use it for obfuscation reasons.

For instance, the example in Table 45 (see 15.1.4) shows a simple calculation within a function that is printed to the console. The corresponding decompiled version (Table 46) does allow picking up the rough function structure. However, there are still some errors, and it can be assumed that the protection is already better than using Java code, where calculations and reengineered code might be obfuscated, but logically always completely correct.

10.3.2 Native code example using the Android NDK

A more realistic example for Android using the Android NDK for compilation is shown in the Appendix (see 15.1.3) in Table 43. The example exposes an extreme increase in program code in the decompiled version (Table 44), while the original meaning of that function is not completely recovered by the aforementioned decompiler.

Here, the reason is the used optimization during compilation (cf. parameter -O3) that makes it very difficult for the disassembler and ultimately, the used decompiler to restore the original functionality. The Android NDK compiles source files with that high optimization setting by

[90] Application Binary Interface [359]
[91] https://retdec.com/decompilation-run/

default, if not specified otherwise [275], while there are several additional protection options as outlined next.

10.3.3 Examples for existing native code obfuscation techniques

The idea of protecting code by, e.g., hiding useful names, is also not new, and almost as old as the used programming languages themselves. For instance, the 1st International Obfuscated C Code Contest took place as early as 1984 [276] and since then the idea remains the same: to remove useful information as much as possible, change all of the structure from simple to confusing, and to make it more difficult for humans, while compilers do not care about the complexity level that much, it might cost performance only. Since there are obviously many possibilities for obfuscation, the current section will introduce examples only.

Hiding information and increasing complexity

Currently, there is a quite simple method that is generally known and is in some of the contest's source codes as well [276]. It removes function names and replaces them by random unique letters. This is because they allow attackers to gain a first insight into the functionality of a defined method based on its name.

In terms of Java and Android (cf. JNI) at least the entry function has to follow a specially defined format and libraries not following that name convention cannot be loaded, which turns out to be an issue in our later proposings and each java file/class expects its very own native function calls.

For instance, "Java_de_tum_in_GeoGame_StartGameActivity_getLicenseStatus" is a native function getLicenseStatus() that can only be called within the StartGameActivity class. Declaring the native function in another file resulted in an error during the development of our evaluation apps.

Obfuscator LLVM

In addition to the used -O3 optimization that already works as a soft-obfuscation, tools like "obfuscator-LLVM" [277] may be currently used to also bring obfuscation to native code, too.

The LLVM project was originally founded by the University of Illinois and basically provides a middle layer allowing static or dynamic compilation even during runtime. LLVM is an initialism and not an acronym, even one may assume that the original naming intention was Low Level Virtual Machine in the beginning [278].

Obfuscator-LLVM uses this toolchain (LLVM) while providing control flow flattening, instruction substitutions and bogus control flow as options to obfuscate existing source codes [277]. For instance, control flow flattening increases the code by adding additional branch instructions to the code, while instruction substitutions try to represent the same functionality in a more complex way by making it more difficult to recognize, e.g., a calculation. Table 47 and Table 48 in the Appendix (see 15.1.5) show practical examples for some of these methods taken from the website.

An example taken from [279] in Table 49 and Table 50 as shown in the Appendix (see 15.1.6) highlights the effects of obfuscator-LLVM on a decompiled source code, and it can be assumed already that it must be much harder now for decompilers, like the Retro Decompiler, to produce equivalent C source code from an obfuscated binary and the resulted, decompiled code is senseless.

The security benefit by using Obfuscator-LLVM was analyzed by Francis Gabriel from QuarkLabs, which is a cybersecurity company from Paris; [280] explains it in more detail. Gabriel concludes that "[even their] script [that they used for deobfuscation] is a good start, it doesn't and will never break all functions protected by OLLVM [(meaning obfuscator-LLVM)] […] The OLLVM project is really interesting and useful because it shows by the example how to manipulate LLVM in order to build your own obfuscator, which can support several CPU architectures. Compared to commercial closed-source protections, we have seen having access to the source code helps to break protections. But it also shows how strongly obfuscation relies on secrets […] Conversely to what many people believe, code obfuscation is REALLY [sic!] difficult. It is not about forbidding access to the code and data, it is about buying time and thinking ahead of how one will break your layers of protection" [280].

Using JNI to protect Java Function Calls

Since native code is more secure, due to less available references as outlined before, an idea was to analyze code using JNI calls with the chosen decompiler. Table 51 in the Appendix (see 15.1.7) shows an example class called "Account" that offers a function "getUsername" returning a string, while the native code example in Table 52 calls that function by providing the parameters and return values for it in a special low level format[92].

Reviewing the restored code in Table 53 by decompiling the related binary file reveals the great obfuscation that this approach offers already, and without a much deeper understanding of ARM assembly as well as NDK/JNI knowledge the intended functionality may not be obtained at all. It can be assumed that it is very time consuming to trace all memory operations that make no immediate sense to a developer. Other researchers like [100] in our research group share this assumption (statement in his final presentation related to that thesis).

10.3.4 Hooking native code

Hooking native code (shared libraries for Linux/ARM) as produced by the Android NDK would be beneficial for any attackers and the current available options should be reviewed as part of this security analysis section. There are a few solutions available as outlined next for that purpose.

Since there are no official native code protections by major app markets available, this technology is mainly of interest to verify 3rd party research or our own solution in the evaluation chapter of this dissertation and it will be introduced to the simulated attackers (students). Moreover, for the sake of completeness the options are introduced next as well, since they belong to the currently available reengineering tools for native code, and 3rd party developers provided native code solutions as outlined in the related work section already, too.

[92] It may be obtained from a compiled class by using the the tool javap, e.g. javap -s -p example.class

Cydia Substrate

Similar to the Xposed Framework for Java [150], Cydia Substrate also provides a similar functionality with the additional capabilities for Android native code (C/C++ code) [151]. Nevertheless, it was not updated since 2013 [152] and does not work on recent Android versions. Therefore, it is not be reviewed further in terms of this work. Interested readers may find a tutorial for creating so-called substrate modules to be used on older Android versions ("2.3 through 4.3" [151]) at [281].

LD_PRELOAD

Another option for intercepting and overriding functions in native code is to use the LD_PRELOAD directive as introduced in [282]. In this approach the target method of a library is developed with the same function signature and the desired functionality. It needs to be compiled as a shared library and uploaded to the device, while setting LD_PRELOAD for the targeted Android app[93]. The trick is that this library is loaded in advance of all other libraries. A short example that works on modern Android versions is illustrated in the Appendix (see 15.1.8) to replace the sendRequest() method call of the nLVL used by the entry JNI method of our evaluation game. The names of these methods may be obtained by viewing the exported symbols (function names) using the Linux tool nm[94].

Frida

Frida (as introduced in [153]) is a framework to intercept processes of various operating systems including Android. It uses JavaScript and Python.

ARM Inject

ARM Inject is a tool to inject shared libraries into running processes for replacing all kinds of functions to intercept and modify communication or to print out interesting information like variables or encryption keys maybe [283].

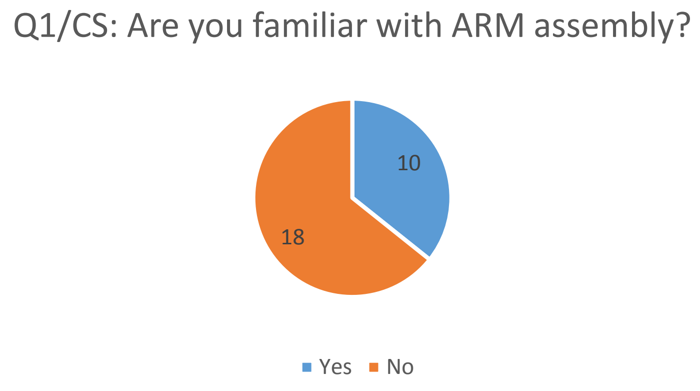
10.3.5 Survey

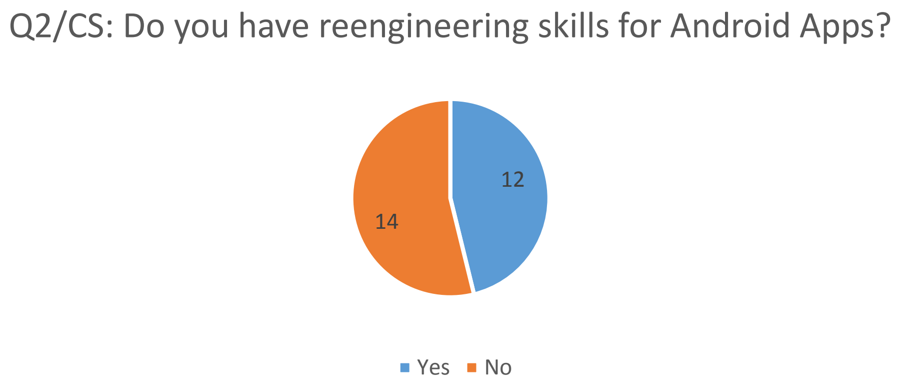
By assuming that typical customers are not familiar with any reengineering techniques, students and skilled developers may know aforementioned tools to disassemble and decompile Android Apps. Nevertheless, we get the impression that many students in related computer science majors are not sufficiently familiar with ARM assembly, and therefore, using native code is already an interesting protection option. In proving this assumption, we conducted a survey on this and other impressions. Having access to our own app and developers, our survey feature was included in the TUM Campus App in June 2016 to provide researchers the ability to ask users simple questions based on their faculty assignment. The survey occurred over a duration of about 14 days. The following diagrams show the results, while the conclusion is presented below and the screenshot of the actual survey is provided in the Appendix (see 15.2.2). App users studying at the TUM Faculty of Computer Science (CS) were asked question Q1 and Q2.

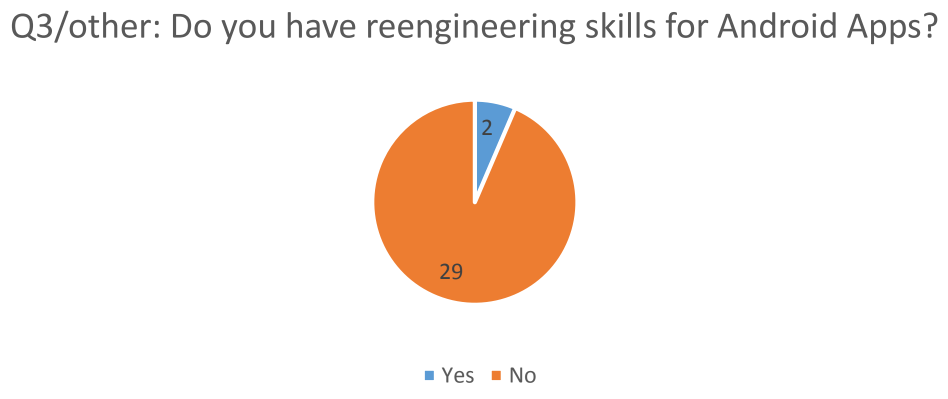
[93] e.g. setprop wrap.de.tum.in.nilsapp LD_PRELOAD=/data/override.so (based on [282])

[94] nm -aDC --defined-only libMyTest.so [281]

Question Q3 addressed non-technical majors (including the faculties for architecture, chemistry, mechanical engineering, medicine, economy, education and sports).







Taking the (non-representative[95]) results into account, they still confirm our assumption that even computer science majors are less familiar with ARM assembly, which is an indicator that such a solution provides more security. We also need to acknowledge that many other students that have a technical major are not familiar with Android reengineering by default. Nevertheless, keeping the reports used in our evaluation in mind (cf. [284] [285] [286]), all students of all skills levels were able to obtain basic reengineering knowledge within the first hours, and tools like the APKtool and others can be found relatively quickly. Ultimately understanding smali language to apply modifications most assuredly requires some existing knowledge in a computer science major. Question Q3 that addressed all other non-technical

[95] due to few participants

majors showed the expected outcome that almost no one is familiar with reengineering Android apps. Assuming that many customers do not have profound knowledge of these matters, then using typical obfuscation might already be safe. Unfortunately, and due to available tools like “Lucky Patcher” [49], even non-professionals are able to circumvent existing and default implemented protections. Therefore, better solutions are a mandatory requirement.

10.3.6 Section conclusion

Using native code is a perfect alternative to protect sensitive code parts that need to be executed in an insecure environment like Android itself because Java code is obviously not secure, and usual obfuscation had almost no effect in our own evaluations (see 13.3 ; cf. visible framework calls), while most developers – even computer science students – are not familiar with ARM assembly (see 10.3.5). Therefore, even trained professionals (= graduated students) will certainly have to spend much more time investigating and obtaining the original functionality than on Java-only solutions. This can solve the critical time requirement of distributors to survive the initial weeks (e.g., about 30 days as suggested by [70]) of a new release without a cracked version being available to have the required earnings.

Native code protection is fairly well researched as highlighted previously (see 10.3.3) and in the proposal section (see 11.5.3). A more complex example using native code to improve the license verification for gaining additional security benefits is in section 11.4.8 about the developed nLVL.

Nevertheless, as outlined in 10.3.4, native code is also vulnerable to attacks and we pointed out improvement ideas for fixing some of these issues in the proposed solutions sections (see 11.5ff). A special issue that arises with this finding is the required combination of Java and native code in a secure manner. That is addressed in 11.4.4, 11.4.5 and 11.4.6 in more detail and developing an app in Java is still the preferred way and much easier than C/C++.

10.4 Existing hardware solutions and comparisons

In recent years, manufacturers started to introduce additional, secured hardware to provide the necessary security level for Android in terms of confidential application data. So far, there are two major technologies available - Secure Elements (SEs) and Trusted Execution Environments (TEEs). Both options are analyzed in this section for their designated purposes of copy protection.

10.4.1 Secure Elements

As briefly introduced in the fundamental section (see 8.5.1), SEs can be programmed by using small Java applets, which are separated from each other due to the virtualization.

Comparing SE to TEE

However, a key difference to TEEs might be an easier and less expensive exchange possibility that would allow for one responsible manufacturer (in terms of copyright protection only), and who is responsible for the card content (cf. usage as dongle). The aforementioned exploit issues, which apply to TEEs, may also be surrounded in that manner, since they apply to SEs as well (see next section).

In an email, Hubertus Grobbel[96] stated further SE advantages when he wrote that even an "SE is exposed to the full range of system attacks. The number of successful attack vectors on smartcards is however close to null." [287] Here, the reasons are "the narrow band / strict interfacing of ISO 7816 and Global Platform standard […] [besides] the highly secured hardware […] that is resistant against DPA/SPA" [97] [287]. Furthermore he confirms the assumption that "An Android host relying on SW security only will never be able to offer […] protection for […] assets like keys […] [and for] processing critical data [. Nevertheless,] End to end security between the two end points of application and the SE is suffering a systematic problem […] [and] valuable data […] available on the Android host […] depends solely on the security of the VM/OS or the application. [However using] […] a secure microSD […] limits the scalability of a successful attack to exactly one endpoint [while] In [sic!] pure SW security a successful attack grants control over the complete system" [287]. Moreover, he stresses that TEEs "are typically wrong marketed promising a secure runtime environment, which is not true in general […] [and its real advantages might be in the field of] 'Secure Display' and 'Secure Keyboard' [suggesting that a combination] […] of TEE and SE is therefore the most […] [promising option realized in a] swedish [sic!] Secure Voice solution […] [by] vendor Sectra" [287].

Possible threats and exploits for SE

Introducing available exploit examples targeting Java Smartcards, they are utilizing, for example, "a known technique of type confusion of the card's Java Virtual Machine by exploiting the faulty transaction mechanism implementation" [288]. It results in access to "arbitrary memory locations on the card" [288], but a requirement is to be able to install the malicious applet on the card first [288] (cf. our previous request was to have one device administrator for full security only).

Other attacks include the muting of the card by executing an endless loop [289]. It's fair to assume that there might be similar exploits on newer cards, too. Nevertheless, Michael Roland, a researcher on SEs from Austria, stated that attacks addressing missing bytecode verification should have been fixed (on existing solutions[98]) by now [290]. Nevertheless, all of the found exploits assume access to the card content. While this might not be possible on the logical side (cf. [291] where "Issuer Security Domain" [291] and "DAP Verification" [291] limit card content modifications to trusted entities), and the available APDU interface offers very few attack surfaces [287], different cards provide different security certifications for hardware.

[96] Hubertus Grobbel is the Head of BU Security at Swissbit AG
[97] "Simple Power Analysis (SPA) and Differential Power Analysis (DPA)" [360]
[98] Added by author

Therefore, hardware attacks are unlikely, too. For instance, the MSC’s chip is “Common Criteria[99] EAL 5+ certified” [192].

In concluding the aforementioned issues, we can fairly assume that SEs provide high level security whenever there is one responsible security domain (= company) only that ensures safe policies for updating and modifying the card content.

SE and its limitations

Unfortunately, SEs need to interact with a host system (e.g., in terms of copyright protection to exchange a key with the more performant Android system) that is most often vulnerable to other exploits (cf. 10.1.2), too. The application side (including drivers) may be reengineered and the actual communication between an app (or driver) and the SE can be intercepted and ultimately understood by an attacker, since the Android assembly code may reveal the codes as long as Android is not used for proxy-purposes only (cf. secure connection to server, see details in 11.6). Therefore, the communication between an application and the SE (maybe inside a MicroSD) may be considered a huge security risk. The actual usage of this exchanged information within the app (cf. Figure 56) is confirmed by [287], too. In contrast, using an Android app that just forwards the information to an external provider (e.g., website) may be considered fairly safe, since that connection may be encrypted, and any modifications can be discovered.

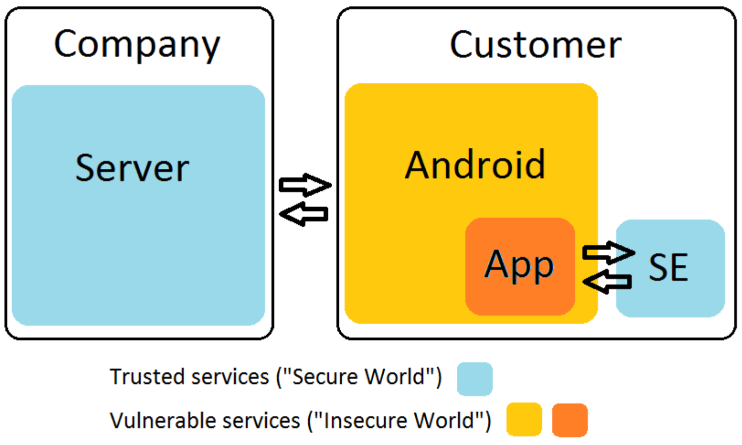

Company
Server
Customer
Android
App
SE
Trusted services ("Secure World")
Vulnerable services ("Insecure World")

Figure 56 - Trusted and untrusted services

This fact is certainly an issue for copyright protection, and at some point, some information must be released to the insecure world of Android, e.g., to decrypt resources that are used in the applications and shown to the user in perhaps the form of graphics or music. The performance of the secure element is not sufficient to perform this task for larger quantities of data (cf. performance section in 8.5.1) in a secure manner and internally.

Another issue known and (sometimes) addressed by PayTV distributors is the so-called “cardsharing [sic!]” [292]. Here, the pirates use a legit card to share encryption keys for PayTV with others over the internet that are valid for a certain time frame only. This issue may be addressed by “behavioural[sic!] contracts” [293], e.g., monitoring the states of an applet and its

[99] Common Criteria (also known as CC) is an international agreement on the requirements of security evaluations [361]

called order, or by expecting requests after a certain time frame while too many requests may indicate a misusage [293].

Of course, in theory, this problem may occur with SEs under Android, since, e.g., any key-exchanges cannot be validated after these keys enter the insecure world of Android (cf. Figure 56) and they may be forwarded to another device to be used illegally there, too (cf. methods that usually require an SE. For instance, temporary access keys for a server provided by the SE). An issue for realizing similar protection is the missing RTC[100] of SEs, and the reoccurrence of attacks in a short amount of time, are difficult to discover.

10.4.2 Trusted Execution Environments

As already presented in the fundamental- and related work section, TEEs have a high potential to provide a secure environment with high performance. Its typical architecture is in Figure 57 and shows the Rich Execution Environment (e.g., Android) on the one side and the TEE with its Trusted Applications (TAs) on the other, as well as possible communication channels. In this section examples for TEEs are reviewed regarding their provided security.

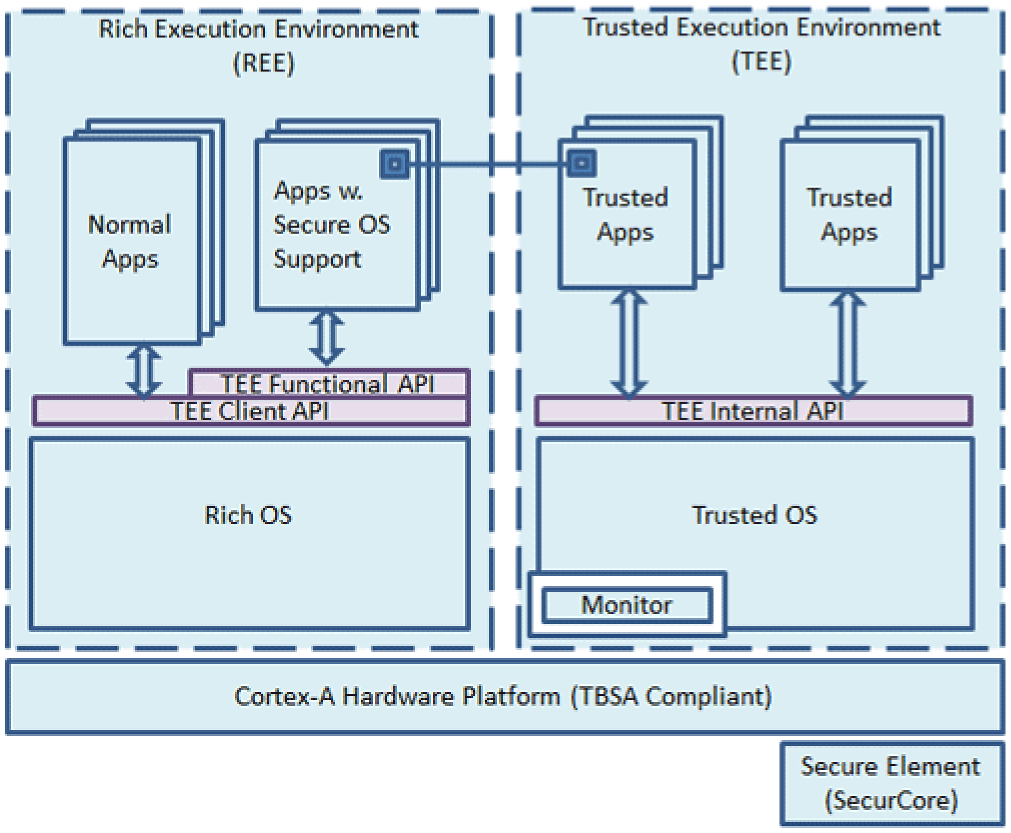

Rich Execution Environment
(REE)
Trusted Execution Environment
(TEE)
Normal Apps
Apps w. Secure OS Support
Trusted Apps
Trusted Apps
TEE Functional API
TEE Client API
TEE Internal API
Rich OS
Trusted OS
Monitor
Cortex-A Hardware Platform (TBSA Compliant)
Secure Element (SecurCore)

Figure 57 - General TEE architecture [294]

QSEE

QSEE is Qualcomm's TEE, which actually refers to Qualcomm's Secure Execution Environment and is executed in SnapDragon processors that are used by lots of current

[100] RTC = real time clock

smartphones [205]. Unfortunately, Atredis[101] [205] found out that several (major) companies are trying to integrate their code into current TEEs like, e.g., QSEE (e.g., Netflix, Disney, etc.). For instance, Atredis stated in a talk about QSEE in 2015 that this reduces the security benefit a lot, since just one of these implementations needs to be hacked to render the whole system insecure. They claim that instead of perhaps providing more security (for banking maybe[102]), TEEs are majorly used for DRM and hiding data from the users. Besides that, TEEs may organize several other features like a SIM unlock, boot loader unlock and protecting the hardware configuration using fuses[103] [205].

In their research [205], they focused on HTC and it turned out that, e.g., in HTC's Qfuses some fuses disabled and others re-enabled functionalities again, which seems not very logical. Also, they claimed that 3rd party companies integrating their code ignored the common Qualcomm specifications and showed insufficient knowledge by leaving the debug code for functions like "tzbsp_oem_do_something" in the production build. They outline that the result is a blackbox with code by many manufacturers and possible security issues. Furthermore, they discovered several architectural issues like missing IOCTL interfaces, no ASLR[104] & DEP[105], an easily cloneable TrustZone image (cf. TEE OS), and physical memory pointers everywhere and so on. Presenting the details, they showed how easy it is on HTC devices to dump the TrustZone operating system to an image file as soon as an attacker got root permissions on Android's system side to act from kernel space. They noted [205] by default that only the Android Kernel was allowed to communicate with the TEE by using secure monitor calls (SMCs) that included OEM's calls like "tzbsp_oem_do_something" and were meant to be available outside the secure world. They revealed that the (TrustZone) image includes a list of all available SMC functions and they were able to discover a write-zero vulnerability in one of the OEM's function calls, which allows overwriting the validation code in the OEM's memcpy function. Of course, that allowed the injection of any code to be executed within the secure world. According to Atredis HTC did not properly reply to the issue. Therefore, TEE solutions by HTC devices in 2015 need to be assumed insecure and not ready for productive use in terms of gaining additional security.

Nevertheless, there are similar approaches for many other devices, including the Nexus 5 and Samsung's Galaxy S5 that also use QSEE. At least in these cases, it is assumed that Qualcomm fixed the issues already [295].

Trusty

Besides aforementioned alternatives, "Trusty" [205] might be a future TEE solution that addresses the improvement ideas by Atredis to release certain codes to the public for verification of the security model. On its website it is still declared "subject to change" [220] (June 2016) and appears unfinished. It is a product by Google.

[101] Security Company www.atredis.com
[102] Added by author
[103] Hardware switches that may be used once to indicate a certain hardware state forever
[104] "Address space layout randomization (ASLR) is a memory-protection process for operating systems (OSes) that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory." [362]
[105] "Data Execution Prevention (DEP) is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system […] [e.g.] execution from data pages" [363]

Trustonic

Another solution in terms of TEEs is the one by Trustonic. Previously introduced in the fundamental section (see 8.5.2), Mr. Ekberg added [203] that a bug in one of the TAs[106] (cf. attacks by Atredis above) would have no effect on other TAs due to separated memory space by different TAs and the TEE OS in general. The reason is that their architecture is similar to user-space and privileged-space on Linux. Nevertheless, Ekberg acknowledged that not all TEEs have this security barrier, and also any bugs within the memory space of the TEE OS would affect other TAs ultimately, too. Right now, their solution does not include ALSR for the 400 million devices available, but they (Trustonic) are continuously improving the product. The solution "Trustonic for Knox can be seen as a strengthened overall solution" [203] that also supports sandboxing of Android Apps for improved security. While a full list of supported devices with their solutions is not available to the public, Ekberg confirmed that it is available in "more or less any Samsung device (phone, tablet) newer than a Samsung S3, newer HTC, Sony, LG (mostly higher-end devices) […] [and for] Android [in most of the] top OEMs" [203].

10.4.3 Section conclusion

From the current point of view, it can be concluded that major TEE solutions like QSEE are not ready yet and further exploits (cf. bad implementation) should be expected until a more secure version becomes available. Moreover, other manufactures like Trustonic are still working on improving their solution and it seems reasonable to wait for the mentioned idea by Atredis of an open-source version that includes available security measures like ASLR, etc. Trusty, as powered by Google, might be a suitable candidate, but is not ready yet either.

Also, due to their high-performance, TEEs are of high interest for future copy protection solutions as illustrated in 11.2.2.

In the meantime, SEs can be used perfectly to bring more security to Android by offering various opportunities, and not only in terms of copy protection. As outlined earlier, even if their performance and data space are limited, they still provide a highly secured environment to perform calculations of encryptions keys or signatures, while any private keys remain secure in the SE itself (in case Android gets rooted). Even Android limits the usage of SEs to insecure Java solutions at the moment (see 11.4.7 for details on the issue), there are still possibilities for securing server access (see 11.4.7), while other solutions do not provide that much more security when realized as a Java solution at the moment (see 11.6ff).

10.5 Overall conclusion

Currently available techniques on Android for software protection and as confirmed in various related student theses (see [75] [64] [66] [100] [62]) and outlined in the previous sections, cannot be considered secure.

[106] Trusted Applications executed in the TEE OS

As outlined previously and analyzed by these different theses, available copy protection mechanisms can be easily circumvented. The issue of an available global cracking tool (Lucky Patcher, cf. [66]) highlights another general problem, too. Developers do not often focus and specialize on this topic and fall back on provided example solutions – the provided sample code by Google maybe. Moreover, in earlier days even obfuscation tools like ProGuard were not enabled by many developers (and unfortunately not by default by Google either), which led to several applications in recent years that may be reengineered easily and even still today. While their figures decrease now, and Google improved the LVL by signed replies [47], Amazon added slight modifications in their DRM version from 2014 to 2016 as well [241]. Unfortunately, the overall issue persists and apps are still easy to reengineer, interceptable (cf. Xposed framework) and are ultimately crackable – in worst case within minutes.

Using the same framework for billions of apps like Googles LVL with few modifications by developers (even Google recommends it [47]) makes matters worse. Moreover, applications lack a secure place to store sensitive information besides secure execution, since they run on Android and store most information within Android's filesystem or its memory (RAM), which is both accessible by root users (assuming the right exploits to deactivate SEAndroid, too).

Nowadays, that issue may be addressed by using SEs and TEEs as it is already used for DRM in terms of media content and its keys. However, it is not used in terms of copy protection for apps yet, and there are not any frameworks designated to allow its usage especially for apps. Instead, there is a DRM framework by Google meant for media content only [296]. It is no surprise that this DRM is realized in native libraries, which provokes the question as to why Google (Android) does not provide similar solutions for apps.

While SEs may provide a secured space and low performance, they are easily exchangeable. Instead, TEEs provide the same performance as the operating system as well as secure space, but are likely to be used by many companies, which may limit their security benefits and increase their attack surfaces, as outlined previously. Both environments may face the issue of exploits, but SEs may be assumed to be more secure when there are responsible policies for managing card content (cf. one administrator, who is allowed to install applets). An issue for SEs, which came up in recent months, is their limitation to be used by (insecure) Java code only, since SEAndroid limits the usage of libUSB and using more secure native code for access is not possible at this moment anymore (for more details see 11.4.7).

Another conclusion that can be drawn is the more secure approach of using native code for apps. While Google introduced the ART VM and allows even performance intensive tasks to be executed smoothly now without native libraries (cf. evaluated in a video app in cooperation with Weptun company in one of our Android practical courses), the embedded DEX code is still a serious issue for hiding protection mechanisms. Therefore, using native code (Android NDK) to handle license requests, as well as many app parts, is still a reasonable solution at this time, even it might not be required for performance reasons anymore. Therefore, the upcoming proposals in the next section are based on the assumption of more secure native code and propagate its usage.

11 Proposed solutions

This section covers the possible solutions to improve the current situation of insufficiently secure copyright protection methods on Android as outlined in previous chapters.

Note that these approaches can increase the security only without being able to solve the fundamental and severe issues of Android‘s insecure operating system and its insecure hardware. Additional suggestions in terms of that matter are addressed in 11.2 requiring the cooperation of external entities like Google or device manufacturers.

The proposed methods that target app developers are sometimes based on approaches by other researchers or companies (see information in each section) and often implemented by requests in related student theses. These range from gaining additional security by reimplementing real copy protection towards APK files by injecting user/device details as a requirement for running an app (see 11.4.1 and 11.2.1), to the ported license verification library using native code for additional security instead (see 11.4.8), while proposing methods to glue insecure Java code and more secure native code together besides presenting known protection options (see 11.4.4, 11.4.5, 11.4.6). Moreover, updated methods for known ideas of loading code dynamically with a focus on Android using ART VM are presented (see 11.4.3). In addition, the question towards secure storage space is discussed (see 11.4.2) and remaining options for using SEs are covered (see 11.4.7), before concluding everything in a best solution proposal in 11.7.

As outlined before, the use of native code is highly suggested. In our evaluations (see 13.4ff), the included methods of using a native license verification library (see 11.4.8) besides the proposed methods to bind Java and native code together, proved to be effective, while Java-only protections (see 13.3ff) have been successfully circumvented by the participants.

The use of SEs in combination with dynamic code loading were not evaluated, but it can be assumed - from a logical point of view – that we can increase the security once more as well, since data and code are stored outside the insecure system and are ultimately revealed by dynamic analysis upon runtime only. The reason for leaving it out is addressed in 11.4.7, while conceptual ideas are presented in 11.6ff.

11.1 Proposed approaches in general

During the most recent years of our research five essential approaches evolved in the sum of all available methods to improve the security in general and to prevent app piracy as presented next in more detail for each item.

- Individualism for implementing protections (each app unique)
- Usage of native code for preventing reengineering
- Regular monitoring of cracking solutions to act on it
- Inclusion of security-relevant libraries for trust
- Apply a basic protection (encryption) for files, databases and network traffic

11.1.1 Individualism

As recommended initially by Google [297] [47] in their description about how to integrate the LVL, it has to be noted that customized programming of security features greatly improves the overall security level of each app. For instance, the LVL may be implemented natively (see 11.4.8) and combined with actual (converted) program code to glue the Java app and native code even further together. This way attackers are not able to separate a well-protected native code from the actual app (see details in 11.4.5 and 11.4.6 with related fundamentals presented in 11.4.4), while the native library looks different to other apps. General cracks – like using simple search – and replacing the pattern (as done by Lucky Patcher [66]) cannot work out.

In addition, the copy protection mechanism may use all kinds of devices- or user-attributes for identification purposes in order to enforce the target license (see 11.4.1). In the end, each app can also be based on other attributes (compiled for a different user/device, see idea in 11.2.1). In this case, crackers would be required to develop a crack specially designed for an application, and none of the general available tools (see 10.1.3) would be available to circumvent the used, individual copyright protection. One of the key issue as outlined in earlier chapters is that any DEX code is not sufficiently protected against reengineering attacks, which is the reason for recommending the use of native code as explained next.

11.1.2 Native code

Even Google does not recommend using native code [7] and we experienced in recent projects[107] that even the streaming of HD video files (usually implemented in native code for performance reasons) no longer requires native code implementations on ART VM (cf. ART VM compiles everything to native code anyway). We have to point out that using native code for security purposes is still highly recommended and even confirmed by a Google employee [163]. A major difference is that there is no insecure DEX code available when using native code right away. Moreover, the obfuscation benefit is already clearly visible in earlier analyses (see 10.3ff). The question about using native code to obfuscate Java is not new in the end (cf. [273]), but as outlined previously, it is also not realized in terms of copy protections by major app markets yet, while other researchers also used it for their solutions already.

In addition, programming a whole application in C/C++ might not be the most preferred way in terms of Android programming. However, it certainly is the preferred way for securing the application right now, since all current Android versions are including the insecure (in terms of reengineering) DEX code in installation files (APK files) as well as embedded in the compiled files (OAT files), as outlined earlier in 7.3.8.

Unfortunately, a huge disadvantage is that Android's NDK is much less powerful than their SDK, and many functions that are easily available by calling the required method in the Android SDK (e.g., to receive the username of the currently logged-in user) cannot be called from the NDK without using Java functions through JNI. This is most often due to missing rights. Sometimes it is possible to implement a method in pure C source code instead (e.g., accessing

[107] Weptun App / Android Practical Course WS15/16

accelerometer sensors), but that is quite rare. We analyzed this further in 11.4.1 when reviewing available sensors and other sources for device- and user identification possibilities.

11.1.3 Verify and monitor cracking solutions

While developers may not be able to ultimately prevent the cracking of a protection, they want to verify that available general cracking solutions are not working after an app-release and test their apps frequently for any incidents. One of the most famous tools known by customers is the “Lucky Patcher” [49]. Further details on the internal algorithms of this tool are found in chapter 10.1.3.

11.1.4 Trust own code only

“Better safe than sorry” is a famous quote by an unknown author and highlights the way developers want to use provided, external services. For instance, the Google Play Store may be modified on rooted devices (also called “Modded Play Store” [49]) and also, security related libraries and services like the LVL may frequently return false results.

For instance, our solution of a native LVL library called nLVL [62] (see 11.4.8 for details) avoids the usage of existing services by Google and communicates to the Google servers directly, while bypassing any proxy tools used by most other Android applications (cf. J. Raedle’s report [285]). Unfortunately, it requires several unusual permissions that are a disadvantage and due to the fact that our code is not yet endorsed by Google to receive special access permissions. Instead, Google’s frameworks do not require these permissions and they are permitted to access most information as defined by Google.

Ultimately, it is a question of trust, as to whether or not the developers wants to trust the services provided by the system or provide the app its own frameworks for security reasons. In terms of license verifications and by knowing the issues with a “Modded Play Store” [49] the question needs to receive a positive reply, and it is highly recommended to implement solutions self instead.

Nevertheless, trusting Android frameworks in general, cannot be avoided, since verifying every single function and framework seems unpractical. However, it is possible to prevent certain attacks. For instance, an attack by the Xposed framework can be detected [298] and alarms triggered (see 11.5.2 for details) if these detection methods are not already deactivated by the attacker in advance.

11.1.5 Using of basic protection

As introduced, e.g., in 8.5, the basic encryption of files, databases and any network traffic is suggested to prevent many problems such as emulated servers or cracked apps. Due to the known reengineering issues of Android these methods can buy some time only. Nevertheless combined with other methods, e.g. using native code, this time advantage can be raised highly.

11.2 Stakeholders

In general, we can identify four main stakeholders on the stage that need to improve their collaborations to rectify the current issues on copyright protection. While each one by itself may already strengthen the protection, better cooperation and guidance would result in better copyright protection and other related topics, like data privacy that also depends on similar goals of securing private and confidential (app) data. These four stakeholders are essentially Google itself, the hardware manufacturers, each app developer and ultimately the customer himself, too. For each stakeholder, we propose different possibilities in the following sections with a main focus on developers, as presented in broad detail in section 11.4.

11.2.1 Google

Background information

Google represents the operating system manufacturer that is responsible to maintain system security on a software level and in cooperation with hardware manufacturers on the hardware level. Google is mainly responsible to maintain the system security by providing safe and ultimately exploit-free services. Of course, this is a major task and due to lots of integrated third party code and libraries (e.g., Linux Kernel, OpenSSL, 3rd party drivers, etc.), as well as many developers, it is an almost impossible one. Most assuredly, Google and their open-source community try to maintain system security and solve rising issues. One of the highlights has certainly been the integration of “SELinux in enforcing mode” [207] that prevents access to files even when acting at the level of a root user. Unfortunately, Google did not manage to create a fast way of distributing system updates in earlier times, in order to apply security patches to available devices more quickly. Everyone knows that each carrier in cooperation with manufacturers, created its own Android system with its very own look and feel. It is recommended to separate the Android system itself from those parts that may be customized by carriers. Apparently, with Android 6, such a system was enabled and the “Android Security Patch Level” is viewable in the about-section of Android now. These devices (we only reviewed the Nexus 5) receive security patches every few weeks now. Nevertheless, it remains unknown to us if it is already enabled for other devices and manufactures in general.

Google is not focused on IP protection at the moment

During our investigations, we got the impression that Google focuses on performance at the moment, since the introduction of native code with the ART VM certainly improved Android’s performance. However, thinking about the known issues of Dalvik Bytecode, we can still discover it in most recent Android versions using OAT files (cf. 7.3.8) [100, p. 18].

In addition, other companies (like Amazon, GuardSquare etc.) showed interest in our research. Google was informed, too. Moreover, Google Android’s security team classified severe threats to IP (see 10.1.5 regarding our dynamic, universal crack) as low security issues (see 14.5 and [164]), since it applies to rooted devices only. It needs to be understood that according to figures collected by researchers in Cambridge [260] (see 10.1.2), by the end of 2014 approximately

90% of the screened devices were insecure, while security patches were available to the remaining 10% in theory only. Therefore, rooting a device is more a question of if the user wants to do it. In addition, a recent report confirmed our impression and “Google representatives declined to comment on the Android piracy issue” [28].

Proposal 1 – Native Code and new market implementation

Similar to the idea of watermarking apps for identifying illegally shared app copies [234], it is recommended to provide customers a new Play Store that offers native code versions with embedded device/user details and getting rid of the embedded DEX code in OAT files. This also includes the current format of APK files that adds the platform independent DEX code once more, too.

One of the reasons, and huge advantages, for using APK files is their platform independent format that gets optimized on each device based on the specific hardware properties as introduced before. That is the reason that Android runs on so many different devices, while Apple’s iOS and their apps are optimized and runnable only on a handful of devices. Nevertheless, a possible approach for keeping that benefit and improving security might be the introduction of a new app store that compiles an application for the target platform based on parameters transmitted by the store’s client app and therefore, solves the copyright protection issue right away as outlined next.

The provided native application can not only be optimized for that single customer device, but even compiled with certain properties of that specific hardware or user details and is therefore runnable on that device only [164]. Figure 58 illustrates this approach. The disadvantage of this idea is the CPU and memory intensive operations on the market side, but with raising and dynamic adjustable cloud computing power it should not be an issue at all.

While it would be relatively easy to develop the necessary market application and server-based services for the compilation, it would be required to modify Android itself. Within our research [100] and limited knowledge on the ART VM, as well as it formats, this is an almost impossible task. There is almost no documentation (except for the source codes) and implementing a modified Android that does not require DEX code anymore, cannot be performed without additional help by Google.

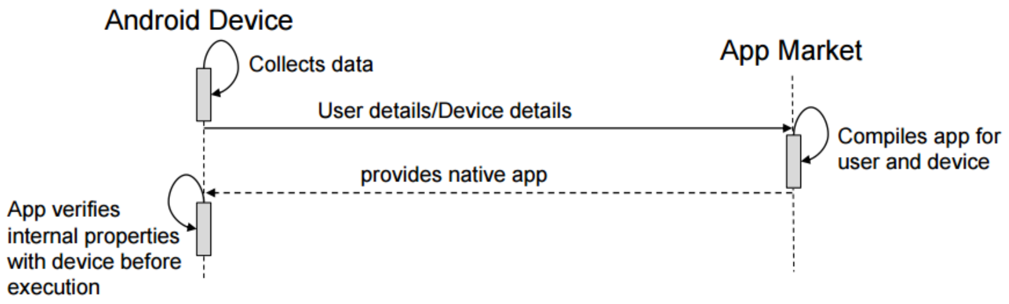

Android Device
Collects data
App Market
User details/Device details
Compiles app for user and device
provides native app
App verifies internal properties with device before execution

Figure 58 - Proposal for new Google Play Store Implementation [164]

As of now, the Android Operating system does not support this approach, and in our research approaches (as performed by [100]), we were not able to get an app running without embedded and available DEX code. The only possibility to satisfy the requirement of DEX code and our idea, is to use a base app that is extended by loading native code dynamically. So, in turn, it would be required to develop the app in pure C/C++. My student and I discussed this whole approach and he came up with similar ideas as well [100, p. 58].

In addition to the proposed idea and as an additional benefit while updating market services, Google may think about integrating certified services for apps as presented by [299] to also provide customers more trustworthy apps.

Proposal 2 – Native License Verification Library and secure local storage

While the above proposal requires heavy changes for Android's ecosystem, there is another possibility to improve current issues with the existing LVL.

The idea is to provide app developers better native access to Android services (cf. many method calls are not available within the NDK, see 11.4.1 for examples) and ultimately an official native version of the LVL not requiring special permissions (cf. current nLVL / see 11.4.8), too.

In a proof-of-concept, the nLVL was developed on request in [62] by us, but due to the fact that it is not being a Google service framework, it requires "strange"[108] permissions to obtain user details like an access token. A protected app is required to obtain these permissions at the moment, of course. The details on the nLVL are presented in 11.4.8 separately and it can already be used by developers (available on request[109]).

Ultimately, Google may pick up that idea to remove the additional, required permissions, while providing app developers a much more secure LVL solution[110] that way.

Furthermore, Google may open up available secure elements to store license information in a more secure manner than in an app's private directory while providing required NDK APIs for access. Figure 59 illustrates the approach in combination with the nLVL introduced above.

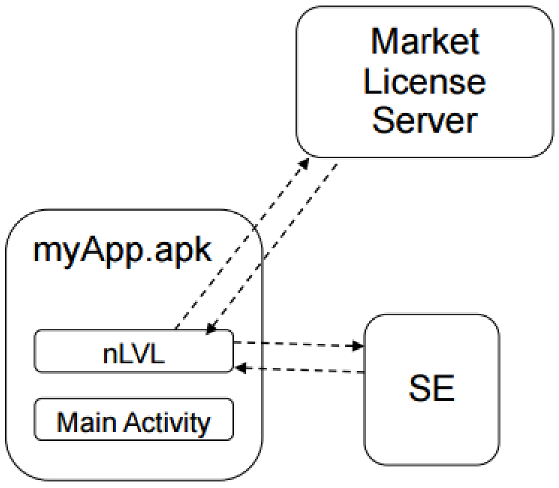


Figure 59 - Native License Verification Library with attached secure storage [164]

Proposal 3 – Port access to user- and device information to the NDK

As outlined in 11.4.1 one of the biggest challenges was to identify available information sources that can be accessed from the NDK and used in native code. It needs to be noted that accessing

[108] from a user's point of view
[109] as of now, the nLVL is not publicly released due to open legal questions (cf. reengineering of Google's interfaces)
[110] assuming to fix left MITM issues by adding the signature validation (see 11.4.8 / known issues)

it via NDK is preferred to hide the actual usage of any functions (sources) of that information even better. While some information could be read from system files or environmental variables using native code, simple information (like the used Google account) are inaccessible and have to be obtained through the Java frameworks using JNI. Here and as a proposal, Google is requested to provide these information within the NDK APIs. Of course, the access needs to be regulated by Android’s permissions to prevent malicious usage and the same ones as for the Java code may apply here, too.

11.2.2 Manufacturers

While the Operating System is provided by Google, the hardware is manufactured by different companies including Motorola, HTC, Samsung, and LG, to mention a few. It is their responsibility to add the required security hardware layers, e.g., disable debugging ports like JTAG[111] for production builds or add fuses (= “one-time programmable bit” [60]) to allow sophisticated security measurements (e.g., blocking of services based on that information as Samsung does [60]), in addition to also providing TEEs with secured space and memory separation on a hardware level.

Currently, there are still many different devices available that already offer one or the other feature. For instance, Samsung with “Trustonic for Samsung KNOX” [202] as introduced in the related work section (see 9.1), now provides an interesting approach by prohibiting the usage of secure containers for business applications. Here, when a device’s bootloader gets unlocked, a fuse bit is used that can be set “from 0x0 to 0x1 (i.e. burned)” [60] once, marking the device forever.

The question is why such an unlocked bootloader is rated a threat, and if there are no possibilities to maintain security, even a custom OS may get installed. While many current devices suffer an immediate security risk once the bootloader is unlocked (and a custom OS usually with available root rights for users installed), it would be recommended to either improve and introduce TEEs even further, or lock the devices down.

In terms of copy protection a more strict policy is preferred (cf. “copy-protection is always to some extent security by obscurity” [1]) and the techniques are mostly available:

For instance, enabling secure boot (also known as “verified boot” [54]) may be a technique to protect a device’s software (bootloader, boot partition, etc.) from modifications and to make sure that only validated ones can be executed. Here, “each stage […] [can verify] the integrity and authenticity of the next stage before executing it” [54]. Nevertheless, thinking about OS updates, this would require cooperation and ultimately standardization between Google and the device manufacturers.

In terms of users’ requests for their freedom in choosing the preferred operating system, TEEs may hold the key to protect apps, critical business data, and ultimately sensitive license information, too. For instance, trusted and checked apps may be downloaded and executed within the TEE (which needs to be Android in our approach) and literally streamed [164] to the

[111] JTAG is a programming & debugging interface for embedded hardware like processors, FPGAs etc. [364]. Of course, still available ports in productive hardware are a perfect interface for attacks

user in his insecure OS, who is able to control an app by sending input signals. Basically the user watches the rendered UI output in that case only and does not own the app to manipulate or copy it in any way. We assume this provides maximum protection. A possible implementation approach might be that the TEE must have direct access to the framebuffer so that a user might expect a smooth execution. While this approach might be more user-friendly, it maintains the copyright holder's requirements for protecting of their IP/apps, too. Of course, it requires new hardware and new TEE OS based on Android. Currently available TEEs can be used to outsource parts only.

11.2.3 Developers

Ultimately, app developers are in charge of protecting their apps in general (as stated by Google as well [47]) and to implement better solutions to existing protections that are preferably even useable across different markets. Section 11.4 covers several proposals that can be used to improve copyright protection as well as license verification.

It needs to be firmly restated (as mentioned previously in 11.1) that the key to successful protection is customization, and the available cracking tools usually only target default implementations. Therefore, having a different protection scheme in each app, makes it much more difficult for attackers and increases the time until a pirated version becomes available. (Note that LP can offer custom recipes to provide cracks for those apps through [66]).

11.2.4 User (customer)

Besides the aforementioned stakeholders, the user himself is another actor, who is asked to only use the services in the designated way, of course. Furthermore, the customer himself offers various information sources, like fingerprints, that may be used for identification. Also, the behavior of using the device or the used Google account can be used to improve copy protection methods, too.

11.3 License options for mobile apps

While there are certainly endless options for software licensing taking the amount of devices or other factors in mind, licenses for Android apps can be classified in only a few categories. Most vendors apply the one device/one user license option that most often implies one device/many users (cf. a user and his family).

One Device, One User

One option might be to limit the app usage to a single device and a single user on that device. Current solutions by the major app markets are not using this option.

Many Devices, One User

Having multiple devices and one user only, it is the preferred license model used by the major app markets like Google Play Store and Amazon AppStore. Customers register an account that they use for downloading, installing and actual usage of an app. In addition, they are able to use their account across several devices to use the apps there, too.

One Device, Many Users

Another option is to allow customers the usage of their app on one device only, while it may be used by an unlimited amount of people, e.g., the customer's family. In reality this license is not officially used, but indirectly used by the customers, since it is assumed that most of them are only using one account per device.

Many Devices, Many Users

It might be interesting to allow customers to use an app on several devices, while also permitting the usage of an unlimited amount of customers. For instance, this option may be used in conjunction with subscriptions and companies, who charge users different amounts based on the amount of devices (volume licenses). This license option is not addressed by Google nor Amazon yet, but can be realized using the proposed methods of the following chapters and becomes important when Android emerges to desktop computers with recent releases like "RemixOS" [300] in the near future.

11.4 Improving copyright protection (developers)

One of the key features of copyright protection for mobile applications must be to enforce the desired license by the developer, e.g., that one app may be used on the same device by an unlimited amount of users, or to limit the usage to a single person maybe instead.

For instance, Google and its LVL [47] limit the usage to a person that belongs to an account, but apps may be used across devices and even by other users (e.g., relatives), who know the required credentials for that specific Google account. Of course, as seen earlier, the LVL can be easily circumvented and should be replaced by more secure solutions as explained next.

11.4.1 Identification to enforce target license

The identification of users, digital content or devices is an interesting approach and other researchers analyzed this already (cf. [234], [235] and [225]). According to an article [public online sources as quoted by [301]], Google is also working on providing developers functions to identify users based on voice or behavior. Ultimately, Google's LVL uses several user- and device attributes to validate the license status, too (cf. [62] / 11.4.8).

Besides the usage of a single piece of information, it is recommended to obtain further details and preferably, information that cannot be faked that easily anymore. In a related research work [302], we approached the details of identifying users and devices without any limitations to available rights or accessible APIs, rated the gained information in different criteria, and created a framework for demonstration and evaluation purposes. Moreover, [251] [254] [252] describe methods for identifying devices using various sensors.

Based on the results by [302, pp. 20-57] [251] [254] [252], the information sources that fulfill our minimal goals of being available on any Android device and that do not require root access, need to be examined. This is because root access is not available to the usual customers, while we target solutions for a broad range of (existing) devices, including smartphones and tablets, in general.

In addition, developers need to try to limit the used permissions for the copyright protection to those ones that are gathered for the app anyway, which may also be preferably the most common permissions. For instance, the top-5 permissions that can be used typically were discovered in a report by the PEW Research Center [[303] as quoted in [304, p. 22]]. Their research report is based on 1,041,336 apps analyzed from June to September 2014 and their results of the most commonly used Android permissions are as follows (while indicating the actual usage in brackets (x of 1,041,336 apps)):

- Full network access (83%)
- View network connection (69%)
- Test access to protected storage (54%)
- Modify or delete the contents of your USB storage (54%)
- Read phone status and identity (35%)

Any additionally used permissions increase the risk that users dislike using the app or even uninstalling an app, when they are concerned about the usage of their information. In fact, PEW Research Center [304, p. 12] discovered that 60% of their analyzed app-downloaders did not install an app after noticing a larger requirement of personal information, while for similar reasons, 43% even decided to uninstall it afterwards again.

In addition, we require information resources that deliver the same and unique information over a long period of time (e.g., even present after a factory reset), and preferably in a short amount of time to allow a quick license response. In [302, p. 55ff] several identifiers mentioned below are already classified by their uniqueness, persistence, availability, required time and required permissions. Nevertheless, the ratings were reviewed and redefined by the author of this work for most sources now.

Besides keeping these considerations in mind, there is another requirement for information sources that these sources preferably need to be accessible through the Android NDK. This is because our solution approach should already use native code as outlined in 11.1.2. Nevertheless, due to permission restrictions or complicated native code constructs (no APIs/own low-level implementation), a combination of native and Java code through JNI might be an additional option for executing Java calls in a safe manner as described in 10.3.3.

Furthermore, from a security point of view, methods that are called and return a single string (e.g., a serial number) should be the 2nd choice, since it is much easier to identify and fake these calls than a more complex computation (e.g., some sort of "sensor fingerprinting" [251]) deeply hidden and obfuscated in the (native) code. Of course, an additional prerequisite is that it can offer a good recognition rate, because nobody wants a customer to be branded as a software pirate. Therefore, we require any method to have identification rates of 99% or more.

Summarizing the requirements above, information sources are needed that fulfill the following strong and preferable requirements as shown in Table 4.

Must requirements	**Further (preferable) prerequisites**
• (A) no root permission needed • (B) available on typical Android devices like smartphones and tablets • (C) require typical, most used permissions (see above) only, or none • (D) unique and persistent information over a long period of time (even available after a factory reset) • (E) quick collection (within seconds) • (F) high identification rate (>= 99%)	• (G) can be gathered using the Android NDK • (H) more complex method (> 10 loc[112]) e.g. calculation with several inputs that produce a lot of assembly code (= complicated to understand by the attacker)

Table 4 - Requirements for the copyright identification mechanisms

The aforementioned criteria are transferred into a table in the schema of Table 5 and presented below in detail with each sensor or information source (see Table 6 and all of the following ones) as analyzed by the author based on mentioned sources or examinations.

Information Source			Description of the sensor/information source
A✓ *	B✗ 1)	C 2)	1) explanation for chosen value regarding B 2) * or just a comment ✓ = requirement fulfilled ; ✗ = requirement not fulfilled
D	E	F	Here required Android SDK permissions are mentioned (may not be required for NDK approach; otherwise mentioned)
G	H	Identifies **U**ser or **D**evice	Details regarding Android NDK approach

Table 5 - Default table with descriptions

Rating of sensors for their usage as information sources

SIM			The SIM card offers lots of unique information like the ICCID for identifying a SIM card, IMSI for identifying the user against the network, and the ADN with phone numbers including the owner's [302, p. 23f]. 1) Typically Smartphone only.
A✓	B✗ 1)	C✓	
D✓	E✓	F✓	android.permission.READ_PHONE_STATE [302, p. 23f]
G✗	H✗	U	There is no NDK method to obtain these information other than using JNI.

Table 6- Identification by SIM (generally based on [302, pp. 20-57])

[112] Lines of code

Wireless Networks Hardware			Up to the most recent Android versions it is possible to obtain unique hardware identifiers such as MAC addresses [302, p. 26f]. Due to Google's move to protect each user's privacy this information may not be available on future versions anymore[113] [305].
A✓	B✓	C✓ *	
D✓	E✓	F✓	(*not valid for bluetooth) android.permission.BLUETOOTH android.permission.BLUETOOTH_ADMIN [302, p. 26f]
G✓	H✗	<u>D</u>	Obtain wireless MAC adddress by reading possible (without permissions), e.g. /sys/class/net/wlan0/address

Table 7 - Identification by Wireless Network Hardware IDs (generally based on [302, pp. 20-57])

Wireless Networks' SSIDs			Beside aforementioned information, wireless networks surrounding the device may be used for identification purposes, too. A typical list of configured networks is available without special permissions by using a dynamic broadcast receiver [106] [302, p. 20]. 1) There is the possibility of new networks and any existing ones could be removed by the user.
A✓	B✓	C✓	
D✗ 1)	E✓	F✗ 1)	none
G✗	H✗	<u>U</u>	There is no NDK method to obtain this information other than using JNI.

Table 8 - Identification based on Wireless Networks (generally based on [302, pp. 20-57])

MMC IDs			Besides using the files for identification (see below), unique IDs from memory cards are available [302, p. 28f]. Most devices do not support physical SD cards anymore, while still using MMCs.
A✓	B✓	C✓	
D✓	E✓	F✓	none
G✓	H✗	<u>D</u>	It is possible to obtain a unique[114] id by reading the files (without permission) "/sys/block/mmcblk#/device/cid […] [# = id ; * = wildcard] /sys/class/mmc_host/mmc#/mmc#:*/cid" [302, p. 28f]

Table 9 - Identication based on files and SDcard IDs (generally based on [302, pp. 20-57])

Files			By looking for files that exist for a long period of time identification is possible as well [302, p. 28f] and the locally stored files are unique to a device, while certain changes are possible. (*device reset. Files may not get copied again.)
A✓	B✓	C✓	
D✗*	E✓	F✓	android.permission.READ_EXTERNAL_STORAGE
G✓	H✓	<u>D</u>	Files are accessible from the NDK and can be analyzed, e.g. by taking the oldest ones and store their names or hash values

Table 10 - Identication based on files and SDcard IDs (generally based on [302, pp. 20-57])

[113] Tests reading the raw files like /sys/class/net/eth0/address succeeded on 6.0.1 without root rights (bug?)
[114] Verified using Nexus 7 and 4. Nexus 4 with factory reset and different Android versions. Same IDs.

Accounts			
A✓	B✓	C✗	Android offers a list of used accounts on a device that may be used to identify a user. It’s not a strong requirement [302, p. 24f], but it’s fair to assume that most users have one.
D✓	E✓	F✓	android.permission.GET_ACCOUNT [302, p. 24f] android.permission.AUTHENTICATE_ACCOUNTS [302, p. 24f]
G✗	H✗	U	It is not possible to access the required database file “accounts.db”[115] in /data/system/users/0 without system rights. Therefore the only way to receive this information is using JNI.

Table 11- Identification based on Accounts (generally based on [302, pp. 20-57])

Contacts			
A✓	B✓	C✗	The contacts of each user are unique information to identify a user [302, p. 29ff]. Changes are rare(*), but new ones can be added. Usually they are being synced to other devices by Google, which sometimes takes a while.
D✓ *	E✓	F✓	android.permission.READ_CONTACTS [302, p. 29ff]
G✗	H✓	U	There is no NDK method to obtain this information other than using JNI.

Table 12 - Identification based on Contact information (generally based on [302, pp. 20-57])

Calling Lists			
A✓	B✗ 1)	C✗	The calling lists of each user offer unique information to identify a user [302, p. 29ff] until the next device reset. Furthermore it is not available on tablets. 1) Smartphone only. 2) Upon device reset all information is gone.
D✗ 2)	E✓	F✓	android.permission.READ_CALL_LOG [302, p. 29ff]
G✗	H✓	U	There is no NDK method to obtain this information other than using JNI.

Table 13 - Identifcation based on Calling Lists (generally based on [302, pp. 20-57])

Location			
A✓	B✓	C✗	The positions of a user measured for a longer period of time may identify a user. For instance, his home or place of work. Android offers different option to acquire the position, e.g., via GSM network, WiFi or GPS sensor [302, p. 37ff]. Usually this information is persistent and the user does not change it that often. 1) It requires several days to acquire accurate data. 2) The user may completely change his position during vacation or due to moving.

[115] see Android’s source code file /frameworks/base/services/core/java/com/android/server/accounts/AccountManagerServer.java

D✓	E✗ 1)	F✗ 2)	android.permission.ACCESS_FINE_LOCATION [306]
G✗	H✓	U	There is no NDK method to obtain this information other than using JNI.

Table 14 - Identification based on locations (of a device) (generally based on [302, pp. 20-57])

Music			Similar to other stored files, music files [302, p. 34] and even their meta data may be used for identification purposes. 1) Nevertheless it needs to be emphasized that cloud services push back local music usage. Therefore, we conducted a survey among all majors, and 57 of 86 or 66% (see 15.2.2) indicated to use local music files still. (*device reset. Files may not get copied again.)
A✓	B✓ 1)	C ✓	
D✗*	E✓	F✓	android.permission.READ_EXTERNAL_STORAGE [302, p. 34]
G✓	H✓	U	The music files can be accessed and data extracted with usual NDK calls.

Table 15 - Identification based on music files (generally based on [302, pp. 20-57])

Installed Packages			In theory, there are millions of apps available that allow fine fingerprinting. In fact, many users rather have the most common apps installed. This limits the identification options here [302, p. 31ff]. Instead, date stamps (creation date, etc.) allow the desired fingerprinting. (*device reset. Date stamps will change or apps are not installed anymore.)
A✓	B✓	C ✓	
D✗*	E✓	F✓	No special permissions required [302, p. 31ff], even /data/app is private
G✓	H✓	D	By using the output of the tool “pm list packages -f”, installed apps may be identified, while using “stat” can reveal date stamps for fine fingerprinting.

Table 16 - Identification based on installed applications (generally based on [302, pp. 20-57])

Magneto-/Accelerometer			For measuring a device’s orientation, the accelerometer and magnetometer are most often used. That way, the typical positions throughout the day can be obtained and ultimately used to identify a device or user [302, p. 39f]. 1) It needs to be analyzed for a few days to identify reoccurring patterns. 2) Other users may have similar habits and users can have a busy life resulting in issues recognizing patterns to identify a specific person.
A✓	B✓	C ✓	
D✓	E✗ 1)	F✗ 2)	No special permissions required [302, p. 39f].
G✓	H✓	U	The accelero-/magneto meter is accessible within the NDK (cf. [307]).

Table 17 - Identification based on device orientation using Magnetometer and Accelerometer (generally based on [302, pp. 20-57])

IMEI			
A✓	B✗ 1)	C ✗	The IMEI number is a unique source of information that is available on smartphones only. Unfortunately, it has the disadvantage that it can be modified (*) by the user and there might be more than one IMEI number depending on the available SIM card slots [302, p. 42f]. 1) Smartphones only.
D✓ *	E✓	F✓	android.permission.READ_PHONE_STATE [302, p. 42f].
G✗	H✗	U	There is no NDK method other than JNI to acquire the IMEI.

Table 18 - Identification based on IMEI (generally based on [302, pp. 20-57])

Serial number			
A✓	B✓	C ✓	Even devices may not offer an IMEI number (e.g. tablets), but they at least provide a serial number [302, p. 44].
D✓	E✓	F✓	It can be fetched without special permissions by using reflections from the system properties, since it is a hidden API [302, p. 44] [308].
G✓	H✗	D	The information may be fetched from the output of a getprop command (system properties ro.serialno) [309].

Table 19 - Identification based on device serial number (generally based on [302, pp. 20-57])

Camera / Pixel errors			
A✓	B✓	C ✓ *	Pixel errors arising in camera pictures are very unique identification factors, but they also appear too rarely. They may be discovered in existing or newly taken pictures [310] [302, p. 45ff].
D✓	E✓	F✓	android.permission.CAMERA [302, p. 45ff]
G✓ *	H✓	D	* Existing images may be processed, while accessing the camera is possible using libcamera2ndk [311].

Table 20 - Identification based on specific pixel errors (generally based on [302, pp. 20-57])

Camera / Dark Frames			
A✓	B✓	C ✗	Dark frames based on dark current (also called "fixed pattern noise" [255]) may be used for identification purposes and provide unique patterns. A huge disadvantage is that the photos must be taken in darkness [255] [302, p. 47ff] [312].
D✓	E✗*	F✓	android.permission.CAMERA [302, p. 47ff]
G✓ *	H✓	D	* Since it can be assumed that no images are available to fulfil the above condition, the camera needs to be used by libcamera2ndk [311], while using the light sensor to detect darkness [302, p. 47ff].

Table 21 - Identiication based on Dark Frames (generally based on [302, pp. 20-57])

Camera / PNU			[254] [256] investigated the so called pixel nonuniformity (PNU) and photo-response non-uniformity (PRNU) noise in camera sensors that has its origin in “different sensitivity of pixels to light […] [due to] imperfections” [254]. It can be extracted from existing images and under daylight conditions that had an identification rate of 90.8% in tests by [256].
A✓	B✓	C ✓ *	
D✓	E✓	F✗	android.permission.CAMERA
G✓ *	H✓	<u>D</u>	*The method may use existing images or access the camera via libcamera2ndk [311].

Table 22 - Identification based on PNU [254]

ANDROID_ID			This ID number is a randomly generated 64-bit number called “Android_ID” [175]. Unfortunately newer Android versions generate such an ID per device-user, but also make this method less attractive/more complicated [302, p. 50f]. * Device reset triggers a new number generation.
A✓	B✓	C ✓	
D✗*	E✓	F✓	none [302, p. 50f]
G✗	H✗	<u>D</u>	There is no NDK method to obtain the Android_ID and JNI must be used.

Table 23 - Identification based on the Android ID (generally based on [302, pp. 20-57])

GSF ID			The GSF ID, Google Service Framework ID, is uniquely created upon first usage of the Google Service Framework. The ID remains the same as long as there is no factory-reset initiated or the data reset by the framework (*) [302, p. 51].
A✓	B✓	C ✓	
D✗*	E✓	F✓	none [302, p. 51]
G✗	H✗	<u>D</u>	There is no NDK method to obtain the GSF ID and JNI must be used.

Table 24 - Identification based on the GSF ID (generally based on [302, pp. 20-57])

Microphone (env.)			The microphone may be used to identify the environment surrounding the device or in other words, the user’s typical environment. Storing all sound events may produce too much data, but storing sound levels every few hours may allow gathering unique ident.-data (e.g., computer fans) [302, p. 51ff]. 1) Environment sounds may change often. 2) The collection of reoccurring patterns takes several days. 3) Background noises may disturb results.
A✓	B✓	C✗	
D✗ 1)	E✗ 2)	F✗ 3)	android.permission.RECORD_AUDIO [302, p. 51ff]

G✓	H✓	U	It is possible to record audio using OpenSL as supported by the NDK [313].

Table 25 - Identification based on the device's environment (generally based on [302, pp. 20-57])

Microphone/Speaker			
A✓	B✓	C✗	As outlined in [251] another option for identifying unique device patterns is the usage of the speaker and microphone. Nevertheless they mentioned it requires a quiet environment in addition to being dependent on the used surface. *High identification rate of 95%. 1) It is required to wait for quite audio conditions.
D✓	E✗ 1)	F✗*	android.permission.RECORD_AUDIO [251] android.permission.MODIFY_AUDIO_SETTINGS [251]
G✓	H✓	D	It is possible to record audio using OpenSL as supported by the NDK [313].

Table 26 - Identification based on device specific properties of speaker and microphone [251]

User's Fingerprint			
A✓	B✗	C✗	Modern devices may provide a fingerprinting sensor that can be used in Android 6.0+ for user authentication [314].
D✓	E✓	F✓	android.permission.USE_FINGERPRINT [314]
G✗	H✗	U	There is no NDK method to obtain the fingerprint and JNI must be used.

Table 27 - Identification based on the user's fingerprint [314]

User's Face			
A✓	B✓	C✗	While using the face to unlock a phone is a common feature on Android 4.0+ [315], we propose to use a movie asking the user to guide the camera, e.g., from his right ear to his nose to take additional biometrical data from many frames for ident.-purposes that are not that easily faked like a single picture.
D✓	E✗	F✓	android.permission.CAMERA
G✗	H✓	U	There is not any NDK method to obtain that live video data and JNI must be used to record the video(s).

Table 28 - Identification based on the user's face

Accelerometer (Vibration)			
A✓	B✓	C✓	In [252], Sanorita Dey et al. propose to use a device's vibration in combination with the accelerometer for identification purposes. While the recognition rate is very high, letting a device vibrate for 30 seconds or more does not seem applicable for most use cases.
D✓	E✗	F✓	none

G✓	H✓	<u>D</u>	The accelero-/magneto meter is accessible within the NDK (cf. [307]).

Table 29 - Identification based on the vibration measured using the accelerometer [252]

Summary of the rating of sensors for their usage as information sources

The following table shows an overview on the available information sources rated to our defined criteria as outlined earlier in more detail.

Source	A[116]	B[117]	C[118]	D[119]	E[120]	F[121]	G[122]	H[123]	De/Us[124]
SIM	✓	✗	✓	✓	✓	✓	✗	✗	Us
Wi.Ne.'s HW	✓	✓	✓	✓	✓	✓	✓	✗	De
WN's SSID	✓	✓	✓	✗	✓	✗	✗	✗	Us
MMC IDs	✓	✓	✓	✓	✓	✓	✓	✗	De
Files	✓	✓	✓	✗	✓	✓	✓	✓	De
Accounts	✓	✓	✗	✓	✓	✓	✗	✗	Us
Contacts	✓	✓	✗	✓	✓	✓	✗	✓	Us
Calling Lists	✓	✗	✗	✗	✓	✓	✗	✓	Us
Location	✓	✓	✗	✓	✗	✗	✗	✓	Us
Music	✓	✓	✓	✗	✓	✓	✓	✓	Us
Ins. Packages	✓	✓	✓	✗	✓	✓	✓	✓	De
Magn./Accel.	✓	✓	✓	✓	✗	✗	✓	✓	Us
IMEI	✓	✗	✗	✓	✓	✓	✗	✗	Us
Serialnumber	✓	✓	✓	✓	✓	✓	✓	✗	De
Cam.PixelErr.	✓	✓	✓	✓	✓	✓	✓	✓	De
Cam. Dark.F.	✓	✓	✗	✓	✗	✓	✓	✓	De
Cam. PNU	✓	✓	✓	✓	✓	✗	✓	✓	De
Android ID	✓	✓	✓	✗	✓	✓	✗	✗	De
GSF ID	✓	✓	✓	✗	✓	✓	✗	✗	De
Mic. (env.)	✓	✓	✗	✗	✗	✗	✓	✓	Us
Mic./Speak.	✓	✓	✗	✓	✗	✗	✓	✓	De
Fingerprint	✓	✗	✗	✓	✓	✓	✗	✗	Us
Face	✓	✓	✗	✓	✗	✓	✗	✓	Us
Acc. (Vibra.)	✓	✓	✓	✓	✗	✓	✓	✓	De

Table 30 - Overview of information sources rated according to our criteria as outlined in Table 4 (based on aforementioned sources)

While options A-F are strong requirements, option G is required for our proposed solution approach (cf. native code). Nevertheless, developers may decide to use a certain source anyway, even though using the Java code version (insecure) and the JNI approach (more secure) are the only ones available. It has to be stated that only native code offers the best protection as outlined

[116] No root permission needed
[117] Available on typical Android devices
[118] Requires typical, most used permissions
[119] Unique and persistent (cf. survives factory reset)
[120] Quickly collectable (seconds)
[121] High identification rate (>=99%)
[122] Android NDK option
[123] More complex method (cf. obfuscation by amount of code)
[124] Information source may be used for recognizing devices (De) or Users (Us)

earlier. Furthermore, option H (cf. complex method) may provide additional security due to the complicated fetching of the information that once again increases the assembly code.

In addition, it needs to be noted that most developers have issues with pure assembly sources due to missing references and pure moving of values in memory, which is difficult to understand in full detail. In addition, it needs to be highlighted that a majority of technical major students is likely not having the required ARM assembly skills (see 10.3.5). If not them, then who else should have these skills? Of course, there are decompilers that produce C source code (see 10.3.1), but native code can be much better obfuscated than Java source code (see 10.3.3, 11.5.3) and the resulting decompiled code might not be helpful.

Applying all these filters (A-G, H optional) to the aforementioned sources, a developer can use the following options for device- and user-identification and for the proposed native solution approach in general.

Source	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**De/Us**
Wi.Ne.'s HW	✓	✓	✓	✓	✓	✓	✓	✗	De
MMC IDs	✓	✓	✓	✓	✓	✓	✓	✗	De
Serial number	✓	✓	✓	✓	✓	✓	✓	✗	De
Cam. Pixel Err.[125]	✓	✓	✓	✓	✓	✓	✓	✓	De

Table 31 - Remaining identification sources providing the best security benefit (based on aforementioned sources)

A visible issue becomes the user identification that depends on information that is non-accessible by the Android NDK (cf. proposal to Google in 11.2.1) or identification methods that are too weak and do not fulfill our high identification rate criteria.

Furthermore, several sources - having less preferred recognition rates - may be combined to achieve a recognition based on probability, but this implies the risk of false identifications. Therefore, it is recommended to rely on the endorsed sources instead.

If user identification is mandatory for a target app license, there is no other option than picking one of the less securable methods on Table 30, while using JNI in native code instead of the actual Java version for at least improved security. It needs to once again be strongly mentioned that developers can choose different sources from Table 30, but this may decrease the security benefits since these methods cannot be hidden as well as the ones in Table 31.

Based on the results of Table 31, two of these methods were selected for the actual implementation in the 3rd evaluation app (see 12.2.3 and its implementation in 15.1.19); however it is up to the developers to choose the amount of implemented and desired methods themselves with regard to the chosen license model (moreover the customization aspect needs to be honored).

In theory, each additional used method can increase the security thinking about the fact that an attacker needs more time to understand it. Ultimately, it highly depends on the implementation and developer skills. In addition, if possible, different recognitions and protection methods should be separated from each other. For instance, instead of collecting the results of all sources to make a decision, the implementations could be separated from each other in different modules of an app and act independently, thus making it difficult for any attacker to find all of

[125] Unfortunately pixel errors are unlikely, but if they exist it is a 100% identification factor

them. This requirement counteracts common software engineering guidelines to sort everything well, but is highly recommended to archive more protection. Methods confusing a developer will certainly confuse attackers, while developers can use comments in the source code to help them keeping the overview.

11.4.2 Requesting and storing of information in a more secure manner

While the identification attributes of Table 31 may be obtained in a more secure manner by using the Android NDK (C/C++ sources), the questions remains where to store this critical information, since most local storage options are considered unsafe:

For example, the external storage[126] is accessible by any app holding the required Android permission. However, the private app space[127] is protected (cf. access rights for app only) and may be accessed only in case the phone is rooted by the user. While rooting causes several issues in general (as outlined in 10.1.2), external sources may provide additional security as Google recommended [316] years ago, too. Unfortunately, using a server - as proposed by Google - would limit the usage of an app by the user (cf. areas without internet connection / flights are not covered). Here, an interesting alternative can be the usage of SEs to fill the gap at least partly[128].

Usage of a SE

By using a SE the information can still be stored locally and permanently attached, but inaccessible to a rooted phone and its user at first view. It would require the reengineering of the application to obtain the access code for the SE and ultimately its stored data besides reengineering the used protocol (we implemented our own file storage and protocol in [197]). Depending on the Android version, the actual implementation and enabled SEAndroid the access code may be hidden in (insecure) Java code or (more secure) native code (see 11.4.7 for details on that issue).

Of course, an additional, remaining issue is that all information exchanged and used on Android may be intercepted. Therefore, we looked for further protection measures in terms of Android. Other researchers in the desktop world already focused on that issue, or related ones, e.g., memdlopen() [317]. This is described by its author as a proof-of-concept for dynamically loading and executing code from memory on x86_64 systems. The reason for its implementation is that the usual system function dlopen() requires an input file[129] that has to be considered unsafe (file easily accessible by an attacker).

In addition, the combination with an approach that uses a SE, as outlined above, allows the setup of additional defense layers, while certain issues, e.g., signing small amounts of data, may be even highly secured, since keys used for signing are not required to leave their secured space on the SE ever. 11.4.7 outlines an idea for accessing servers more securely in more detail.

[126] /sdcard/ (on Android)
[127] e.g. /data/data/com.example.test
[128] not all data/logic can be stored on the SE due to its performance and general limitations (e.g., other databases, etc.). It can act more like a mirror of it.
[129] cf. man dlopen

As already outlined in the related work section (see 9.1), even Google is still internally investigating the options in their project with the codename “Vault” [211]; besides, there is the fact that many devices use SEs, but those are so far inaccessible to default app developers to install on their own applets (see 8.5.1).

11.4.3 Dynamic code loading

In a seminar paper by Mr. Hugenroth et al. [226] and Mr. Schulz [266, p. 14], ideas for dynamic code manipulation were previously discussed. In this section, we now propose and verify some of their approaches in a modified way for Android. If not mentioned otherwise, the presented possibilities are usable under the ART VM.

Dynamic Java Code loading

Originally introduced to Android by Google and hidden in its source code (see Androids Open Source project) as outlined in [64, p. 78f] as well as [266], and then reanalyzed for modern Android versions in [100, p. 29ff] (based on [266]), it was possible on former Android versions to load executable DEX Bytecode provided dynamically. Either from any source in a byte array by using reflections for these private methods, or by calling them from native code using system call dlopen() and libdvm.so as the target library. An issue on modern Android versions using ART VM is that these calls were removed from Android, since the ART VM cannot execute Dalvik Bytecode anymore [100, p. 31].

The remaining public function executing Dalvik Bytecode dynamically is only possible by providing a DEX file stored on the file system (very insecure) that gets compiled by Android on-the-fly, while being stored directly again [100, p. 30ff]. An example taken from [100, p. 30ff] is shown in Figure 82 in the Appendix (see 15.1.9). Here, the DexClassLoader is initialized with two files – one representing the actual DEX file and the second one the actual file that gets overwritten with the compiled version in OAT format. The dynamically loaded class as shown in Figure 81 in the Appendix (see 15.1.9) provides a simple multiplication method that can now be invoked. The whole process comes with a decrease in performance due to the compilation procedures required by ART VM. The required time depends on the actual size of the DEX file [100, p. 31].

Since all input and output files are in control of the app developer, the method may already be considered more secure, even though the temporary existence of these files provides an unwanted attack surface. Former Android versions apparently [266, p. 11] even created an optimized version in the known system folder. However, this is inaccessible due to missing permissions providing an additional threat on these versions. This was likely the reason for using and calling the removed functions in former security solutions that required a byte array of DEX code only (= memory-only solution; see related work section, e.g., 9.2.4).

If an attacker gets access to the dynamically created files, the security benefit decreases completely and the Dalvik Bytecode may be decompiled to easily-understandable smali or even Java Code, as outline in 10.1.4.

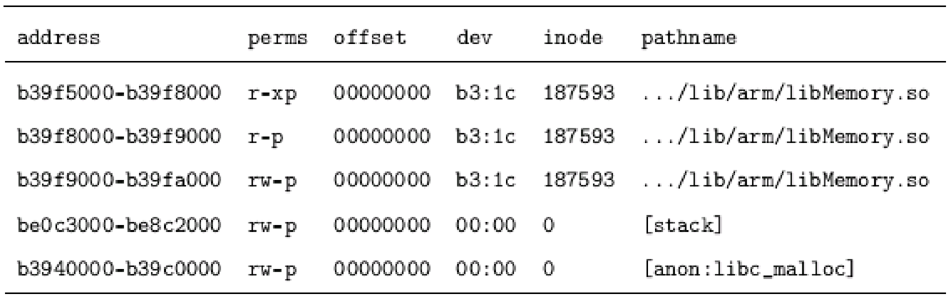
Dynamic loading of Native Code (C/C++) from a file

Similar to the presented method of dynamic code loading in Java code, the same also exists for native code by using systemcall dlopen(). The example taken from [100, p. 35ff] in the Appendix (see 15.1.10) shows the possibility to load a native shared library. It requires as a parameter the native code that will be loaded by dlopen(). Using systemcall dlsym(), the actual symbol name (function name) needs to be found, before it can be invoked. A possible input file is the (compiled version) of the code shown in Figure 84.

Dynamic loading of Native Code (C/C++) from memory

Fortunately, the old and better method for loading code from memory still exists with the limitation that it might only be used momentarily for smaller pieces of native code. Based on the initial idea by [317] [266], we evaluated the possibilities on Android to create a similar function to load native assembly code on-the-fly [100, p. 41ff].

For recognizing how this is done, some additional Linux knowledge is required regarding the memory management of processes on Linux. As explained in [318] Linux (used by Android), maps different parts of an app to various addresses in memory. This information may be obtained in detail by viewing /proc/$PID/maps that reveals a table, as shown for demonstration purposes, in Figure 60. The first column represents the memory region, where that specified section (see column pathname) is stored. The perms (permission) column describes the access rights of that region. Besides known possibilities such as read/write/execute, a region may be shared with other processes (indicated by s) or defined as private (p). Illegal access results in a segmentation fault error. Systemcall mprotect() may be used to modify these settings. The third column represents the file offset, where the mapping started (in case a file was used), otherwise it remains just 0. The fourth column shows the device number (in case of a file), while the fifth one represents the inode of that file. The last column shows the actual path in addition to special names like [heap] or [stack]. Another specialty is a blank field in case of anonymous mapped regions that are also used in our approach.



address	perms	offset	dev	inode	pathname
b39f5000-b39f8000	r-xp	00000000	b3:1c	187593	.../lib/arm/libMemory.so
b39f8000-b39f9000	r-p	00000000	b3:1c	187593	.../lib/arm/libMemory.so
b39f9000-b39fa000	rw-p	00000000	b3:1c	187593	.../lib/arm/libMemory.so
be0c3000-be8c2000	rw-p	00000000	00:00	0	[stack]
b3940000-b39c0000	rw-p	00000000	00:00	0	[anon:libc_malloc]

Figure 60 - Memory mapping of a e.g. a process [100, p. 43]

In terms of Android processes the own memory mappings of each process may be read and even modified by the process itself, while reading other processes' memory mappings would require root rights instead [100, p. 42].

The actual code execution is initiated by storing the machine code in memory first, as explained in [100, p. 43ff]. This may be done by using systemcall mmap() as recommended by [319], since it offers some advantages against the famous systemcall malloc() that would require a developer to take care of additional issues, e.g., that "the allocation is aligned at a page boundary [to avoid] […] unwanted effects […] [with systemcall mprotect()] […] enabling/disabling more than actually required" [319].

By default, as mentioned in [100, p. 43], ELF binaries, or shared object files, may be used, and loaded either statically or dynamically, while the linker takes care of the linking of executable data to the main binary. While this approach would be the preferred one in terms of this work, it is not trivial to realize such a functionality and would require modifying existing linker capabilities as previously done in [317] for x86_64 systems in his proof-of-concept.

Therefore, it was decided to look for a different approach that still fulfills our requirements of loading code to enable developers to execute smaller security measures strongly and dynamically hidden, even if it resulted in limitations, since complex function calls may not be realized that way. For instance, functions or library calls cannot be performed without adding these additional (linker) capabilities [100, p. 46].

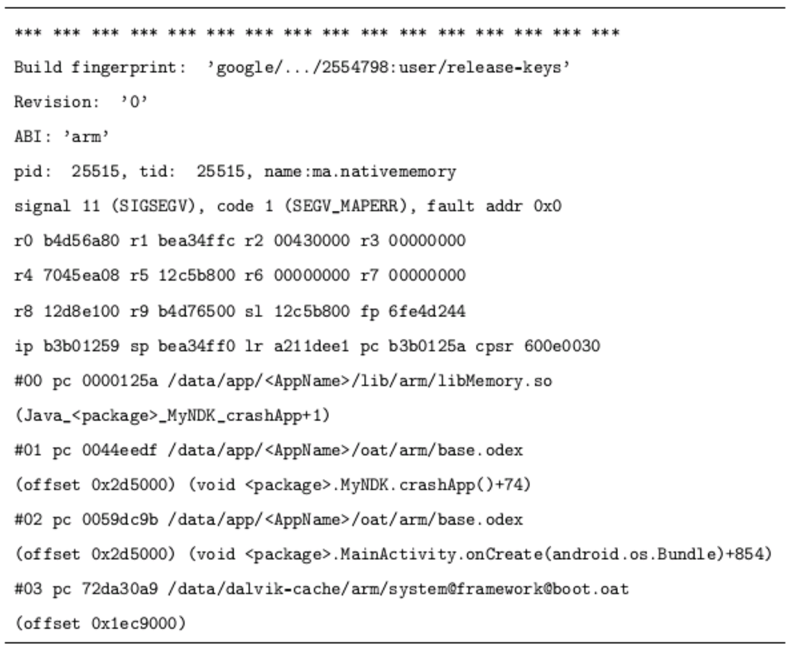
Inspired by ideas from [316] [226] [266] the security ideas provided by dynamic native code here may be endless, and interesting options, which are not limited to memory-only code, might be, e.g.,

- the modification (correction) of statically stored wrong encryption keys within the code to confuse attackers analyzing the app statically, while using the correct keys upon runtime
- decryption of previously encrypted data with a dynamically loaded algorithms from an external source (e.g., SE)
- enabling or disabling code sections (e.g., functions that quit the app if not disabled during runtime) by using so-called eggs for easy identification and modification, e.g. if ("treeCP" == "treeCP") then exit() ; (see 11.4.4 for an example implementation)
- provide a copy protection with interesting actions in case of a license failure, e.g., crash the app by writing to protected areas resulting in a segmentation fault and making it much harder to understand stack traces (see an example in Figure 62) [100, p. 50]. In addition, native code obfuscation may be used to remove any useful information in function names. Since the code is loaded dynamically it will not be possible to track the code without dynamic analysis tools (e.g., debugging using gdb and pure assembly).

```
unsigned char *p = 0x00000000;
*p = 1;
```

Figure 61 - Example code to trigger a native crash (SIGSEGV) [100, p. 50]

Figure 84 in the Appendix (see 15.1.11) shows a small portion of C source code to be used as dynamically loaded code as described in [100, p. 44ff]. This may be based originally on [319], too. By using one of the available cross-compilers, e.g., GCC's one, an object file can be generated using the command "arm-none-eabi-gccc –O3 –c source.c –o output.o". Next the tool objdump can be used to disassemble the binary code to human-readable mnemonics by executing "arm-none-eabi-objdump –D output.o", resulting in a representation as shown in Figure 85 in the Appendix (see 15.1.11). Here, "byte sequence 0xe0000091 represents the raw machine code for the mnemonic equivalent of a multiplication (r0 = r1 * r0) and the following sequence is responsible for exiting the function and returning the value. Since object files are normally Little Endian, the bytes written to memory have to be reordered (0xe0000091 turns to 0x910000e0) before execution." [100, p. 44].



```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint:  'google/.../2554798:user/release-keys'
Revision:  '0'
ABI: 'arm'
pid:  25515, tid:  25515, name:ma.nativememory
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x0
r0 b4d56a80 r1 bea34ffc r2 00430000 r3 00000000
r4 7045ea08 r5 12c5b800 r6 00000000 r7 00000000
r8 12d8e100 r9 b4d76500 sl 12c5b800 fp 6fe4d244
ip b3b01259 sp bea34ff0 lr a211dee1 pc b3b0125a cpsr 600e0030
#00 pc 0000125a /data/app/<AppName>/lib/arm/libMemory.so
(Java_<package>_MyNDK_crashApp+1)
#01 pc 0044eedf /data/app/<AppName>/oat/arm/base.odex
(offset 0x2d5000) (void <package>.MyNDK.crashApp()+74)
#02 pc 0059dc9b /data/app/<AppName>/oat/arm/base.odex
(offset 0x2d5000) (void <package>.MainActivity.onCreate(android.os.Bundle)+854)
#03 pc 72da30a9 /data/dalvik-cache/arm/system@framework@boot.oat
(offset 0x1ec9000)
```

Figure 62 - Resulting native stack traces after execution of code in Figure 61 [100, p. 50]

The next step [100, p. 44f] is to allocate the memory for that byte sequence using systemcall mmap() while setting the required flags for future execution as shown in Figure 86 in the Appendix (see 15.1.11). Since mmap() is not using a file here, the MAP_ANONYMOUS flag needs to be used. For security reasons, the authors of [100, p. 45] [319] recommend the common idea, known from other topics, to use only the required permissions for each step, e.g., to instead permit execution right ahead of the actual execution. This may be realized using the systemcall mprotect().

In addition, the previously created machine code bytes need to be copied to that special memory area now (Appendix (see 15.1.11) / Figure 87 ; hardcoded for demonstration purposes only) and cast to a function pointer using typedef as seen in Figure 88. The code needs to be surrounded by the typical JNI structures to ultimately also be used on Android [100, p. 45f].

11.4.4 Process memory modification

As outlined above, a possible idea for a copy protection might be to place a great deal of code into an app that quits or renders an app inoperable, while deactivating these calls upon successful license verification. Based on [100, p. 51f] a sample application with its related native code function as shown in the Appendix (see 15.1.12) in Table 58 and Table 59 was created. This idea was inspired by similar ideas from [226].

The code in Table 58 shows the comparison of two byte arrays that would be executed by default forcing the application to quit. Nevertheless assuming that the native code in Table 59 was executed previously, which also checks for a valid license in a real-world scenario, the byte arrays get modified dynamically in memory, which again disables the quitting.

For further protection these sort of methods may be used more often so that an attacker cannot remove the protection by eliminating just one of these conditions. Furthermore, many other eggs (cf. “NILS2K”[130]) may be used and even acquired from an external source, such as a SE or generated dynamically upon runtime. For example, a SE may have fetched the required eggs in advance that need modifications for a specific app version, while providing these information to the app (native code) dynamically for even increased security.

A known issue comes with that fact that Android must have loaded the structures to memory to be available and modifiable by the native code. Therefore, the declaration of variables must have taken place before the native code function gets called. For better hiding and usage in other activities that are not loaded at that moment, a singleton pattern can be used as shown in the Appendix (see 15.1.12) in Table 57 while initializing it in the OnCreate() or onStart() method of the MainActivity by “Global.getInstance();” and before the execution of the native code. Basically, it provides Android global-variable functionalities.

11.4.5 Indirect method triggering

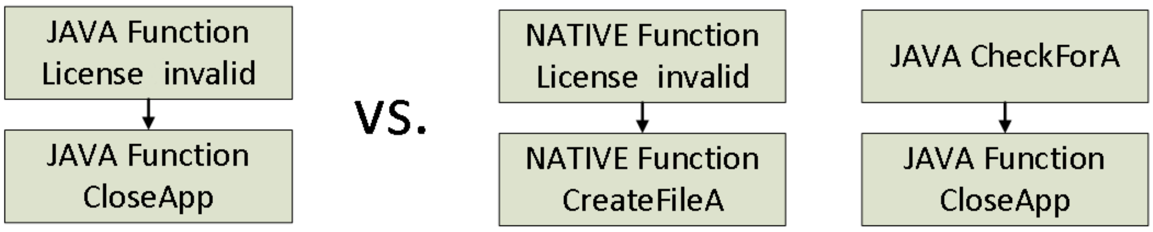

JAVA Function License invalid
JAVA Function CloseApp
VS.
NATIVE Function License invalid
NATIVE Function CreateFileA
JAVA CheckForA
JAVA Function CloseApp

Figure 63 - Illustration of Indirect Method Triggering (example using files)

While any developer usually learns to create applications in a good structure, one of the best obfuscation methods is to ignore some of these guidelines adding, e.g., non-sense code or apparently dead-code that gets called elsewhere.

Indirect communication using files

An interesting combination comes here by using Java and C Native Code that may be called directly or indirectly from each other. While a native copy protection verification method could issue the call to quit the application, it may instead modify some settings elsewhere instead that have no immanent impact (e.g., creating a file; cf. Figure 63), and cannot be traced right away by an attacker for that reason, since no one assumes a relation here. Nevertheless, either a service, or just a function in Java, may watch for such modifications like a created file on the SD-card to either initiate quitting now, or to render the application useless by modifying important app settings that have a negative impact on the user experience[131]. For example, the German company and game-creator BlueByte used such a protection years ago in their famous computer game “The Settlers”. Here, gold mines produced pigs instead of delivering gold nuggets, which prevented gamers (here software pirates) from enjoying the game after already playing it for a while. Moreover, a sequence of these options may be used, e.g., in case of a

[130] Notice the representation as byte array, since Strings cannot be found in memory that easily [100, p.52]
[131] Notice: Here the app is pirated and in our work we assume that pirates can face issues even it may result in negative feedback or comments on gaming websites or forums etc.

negative license reply then a file can be created. Another function detecting that file increases a value, while a different function creates another file elsewhere. This triggers another function at some point that activates different events like app instability or malfunction (e.g., changing game settings that bother the user (here: pirate) such as reducing a players score in a gaming app).

Indirect communication using environment variables

In a similar way the environment variables may be used. They are visible to the actual process and its childs only, and allow interesting options for activating protection measurements in any class and any later program point making it extremely difficult to trace its origin. For example, a native library may set an unsuspicious variable loadTime with value 1 by using the common method setenv(), while it can be read from the actual (Java) app to trigger protection measures, like increasing loading times drastically, modifying resource paths leading to app crashes etc. – there are lots of possibilities and depend on the actual app and the developer’s preference to handle such a possible piracy case.

Indirect communication using dynamically loaded code

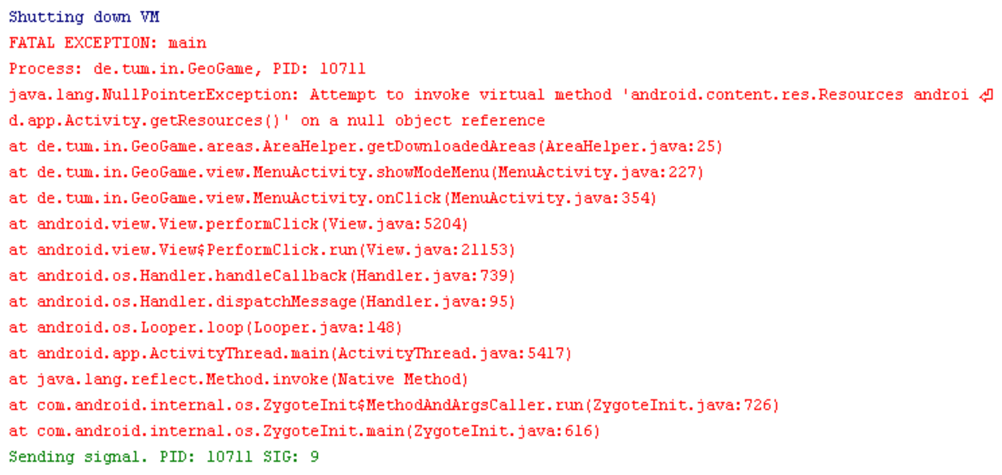
Moreover, dynamically loaded native code from memory introduced previously may be used to modify a variable almost untraceable, and triggering protection mechanisms upon license failure such as crashing the app with that dynamically loaded code (see 11.4.3 for details).

Indirect communication using broadcast receivers / settings

In addition, enabling and disabling hardware (or its settings) within a certain time frame (e.g., GPS) using resulting broadcast messages can be used to exchange information between different program parts to trigger copy protection actions as explained previously.

Indirect app disabling by using killProcess

One of the method calls that also falls under this category is the function killProcess() used to finish the current, or another process, in theory. While one may assume that it instantly kills the process with its displayed activity, it needs to be stated that this is not the case. Instead, by killing the process id of the app, it is only partly shutdown, and that results in strange behaviors. Attackers are not able to identify the actual reason for this erroneous app behavior that makes the app pretty much unusable. Every action results in a crash referring to the NullPointer exception, as shown in Figure 64. There is no indication anywhere that in our program line 145 of startActivityGame.java (example) triggered the whole issue with the execution of killProcess(). Furthermore, it is recommend to execute the kill function in native code, since it is even harder for attackers to identify it and link it to the fatal exception below.



```
Shutting down VM
FATAL EXCEPTION: main
Process: de.tum.in.GeoGame, PID: 10711
java.lang.NullPointerException: Attempt to invoke virtual method 'android.content.res.Resources androi ⏎
d.app.Activity.getResources()' on a null object reference
at de.tum.in.GeoGame.areas.AreaHelper.getDownloadedAreas(AreaHelper.java:25)
at de.tum.in.GeoGame.view.MenuActivity.showModeMenu(MenuActivity.java:227)
at de.tum.in.GeoGame.view.MenuActivity.onClick(MenuActivity.java:354)
at android.view.View.performClick(View.java:5204)
at android.view.View$PerformClick.run(View.java:21153)
at android.os.Handler.handleCallback(Handler.java:739)
at android.os.Handler.dispatchMessage(Handler.java:95)
at android.os.Looper.loop(Looper.java:148)
at android.app.ActivityThread.main(ActivityThread.java:5417)
at java.lang.reflect.Method.invoke(Native Method)
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:726)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:616)
Sending signal. PID: 10711 SIG: 9
```

Figure 64 - Crash log after the process was killed using android.os.Process.killProcess(android.os.Process.myPid());

Impact on the attacker

Using the proposed indirect methods, a static analysis may not reveal the issues for instability right away, and the attacker would be required to identify the long-sequence-tail until finally identifying the actual reasons by using dynamic analyses. Ultimately, it is like playing hide and seek with an attacker. The more complicated and confusing the queue of methods is, the more difficult it becomes for an attacker to reveal and circumvent it; and then, he may not discover the initial reason at all – creating a file is as innocent as turning GPS on and off. Of course, the impact of the methods on other apps must be considered. Using the GPS status for communication, it is recommend to use that resource shortly only to avoid disadvantages for other apps running in background.

11.4.6 Code fusing / fusing options

While assuming that the native code (optionally using an SE for storing data or using encrypted files) takes care of the licensing and the actual app is implemented in Java mainly, a suggested option is to combine (fuse) these different worlds together by converting some of the Java code to C source code and making the native library a mandatory part of the application. In that case, the native code is not only used for license verification, but in addition, a separation from the less protected Java code is not easily possible anymore.

As presented in 11.4.3, 11.4.4, and 11.4.5 in detail the following options belong to this category as well:

- Using environment variables to exchange information about the license status between native code and Java code to take appropriate actions, e.g., set a correct path to mandatory program files in a valid license state only. Obviously, without the native code being executed, the Java application will fail to run now.
- In a similar way, files or Broadcast messages may be used to trigger certain actions on both sides.
- Moreover, the presented option for modifying coded variables dynamically allows interesting implementations that appear to have no sense to the usual developers (cf. comparison of equal byte arrays leading to quitting an app)

These ideas were partly implemented in the 2nd and 3rd evaluation app (see 12.2.2 and 12.2.3 for details), while securely protecting the 3rd application from a separation from the well-secured native code (see full evaluation in 13.4).

11.4.7 Secure Element and its options for copy protection

Unfortunately, the planned solutions for using a SE for improved copyright protection are not possible anymore on targeted Android versions using the ART VM (mainly Android Lollipop, Marshmallow and future ones) in a native code version, which is the desired implementation in terms of security gain.

The reason is the switching of SEAndroid to an enforcing-mode with the Android L release [52], which results in access-issues and related crashes of the used libUSB library – a mandatory library used by our native libaums version (see Appendix 15.2.1 for logging outputs).

Implementing the same ideas in Java code is not an option due to the known security issues related to easy reengineering while having almost no security benefits in that case. Attackers may easily acquire access keys and intercept the communication between the app and a SE. The remaining options for possible solutions using SEs are illustrated below. Even they require placing the general access key[132] in insecure Java code, too.

In the end, it is preferred that global players like Google or device manufacturers enable USB access for certain applications again to allow the usage of native SE solutions or to open up available SEs so others can use them.

Therefore, most ideas are now presented in section 11.6 instead. They cannot be currently realized in a secure manner and are presented as conceptual ideas only.

Remaining option (example)

Nevertheless, there are exceptions, and using an SE for providing unique and temporary access to a server does not depend on a secure native version of libaums library, since the access code (here, a signed timestamp by both entities – server and SE) is meant to be used by Java code anyway, and valid only temporarily, too. Initially, the client app requests a signed timestamp from the server (cf. server has shared secret with SE) that gets validated by the SE using the shared secret. It is then signed once more to allow the server the verification. This approach does not only secure the access to clients owning an SE, but it also allows limiting the reply time by the client to avoid issues with sharing the access option with non-legitimate clients (cf. cardsharing issues [292] and solutions [293]). Figure 65 illustrates this approach in a sequence diagram.

The signature itself can be realized using a timestamp, while appending a secret key - known by the server and SE only - and hashing that concatenated string, while appending the resulting hash-value to the original timestamp divided by a delimiter.

[132] Libaums is used for file IO access, while the SE requires a special, unique key to reply to any requests first

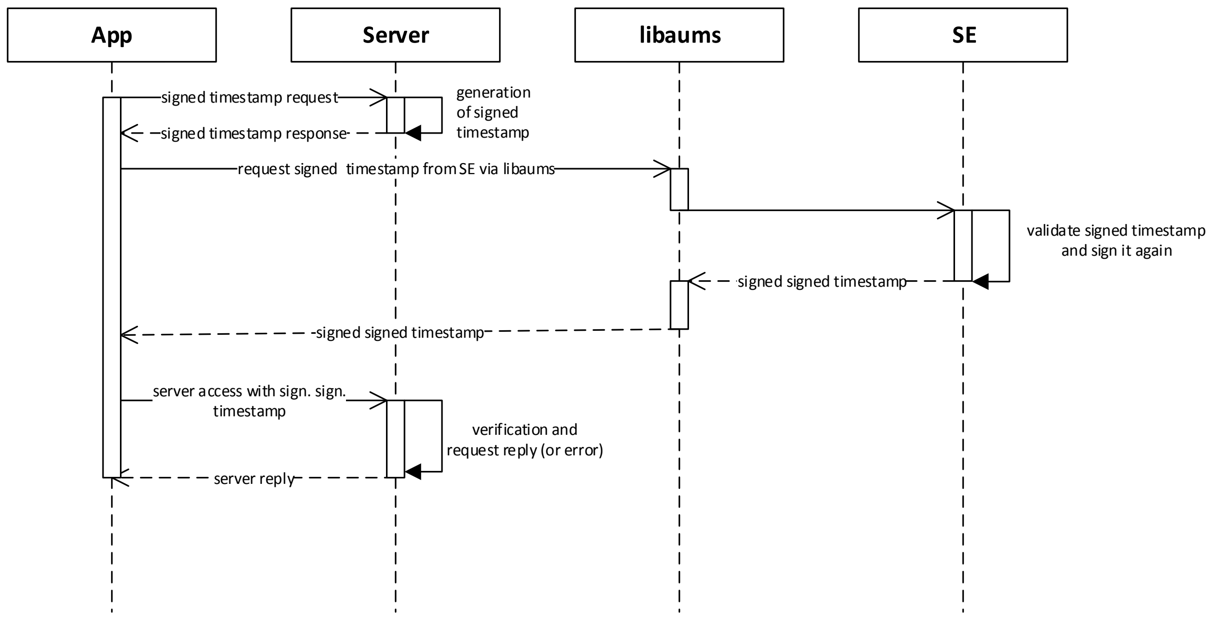

App
Server
libaums
SE
signed timestamp request
generation of signed timestamp
signed timestamp response
request signed timestamp from SE via libaums
validate signed timestamp and sign it again
signed signed timestamp
signed signed timestamp
server access with sign. sign. timestamp
verification and request reply (or error)
server reply

Figure 65 - Possible implementation of a secured server access requiring a SE on the client side

11.4.8 Native license verification library (nLVL)

Based on the discovered issues by others [320] and own investigations performed in [64] as outlined in 10.1.5, an idea arose to port the existing LVL to native code, while solving issues with interceptable and possibly faked services (cf. “Modded Play Store” [49] / 10.1.5) at once, too.

For the implementation of a proof-of-concept of such a native library, the LVL as well as the communication between the Android frameworks and services, and ultimately the Google License servers had to be analyzed (reengineered/intercepted) first, before an implementation focusing on the main functions was possible. The analysis and implementation was performed by request in [62].

The fundamentals on the LVL were already presented in 8.3.2, while even further information on the detailed process using IPC for the communication with the Google services, were presented in 10.1.5. In an even more detailed analysis (for details see [62, p. 26ff]), the actual network communication performed by these services was analyzed. It is arranged on Android by a special network scheduler called “Volley” [321] to prioritize different network requests. For simplification the nLVL implementation focuses on providing the main functionality only and does not use it [62].

As mentioned above, and by circumventing Googles local services (cf. 8.3.2 and 10.1.5), issues with “modded Play Store[s]” [49] may be avoided, and the nLVL library directly now communicates with Googles’ servers by using CURL[133] library as described in great detail in [62, p. 55ff] and shown as the architectural overview in Figure 66 below.

[133] library to allow file transfers - https://curl.haxx.se/libcurl/

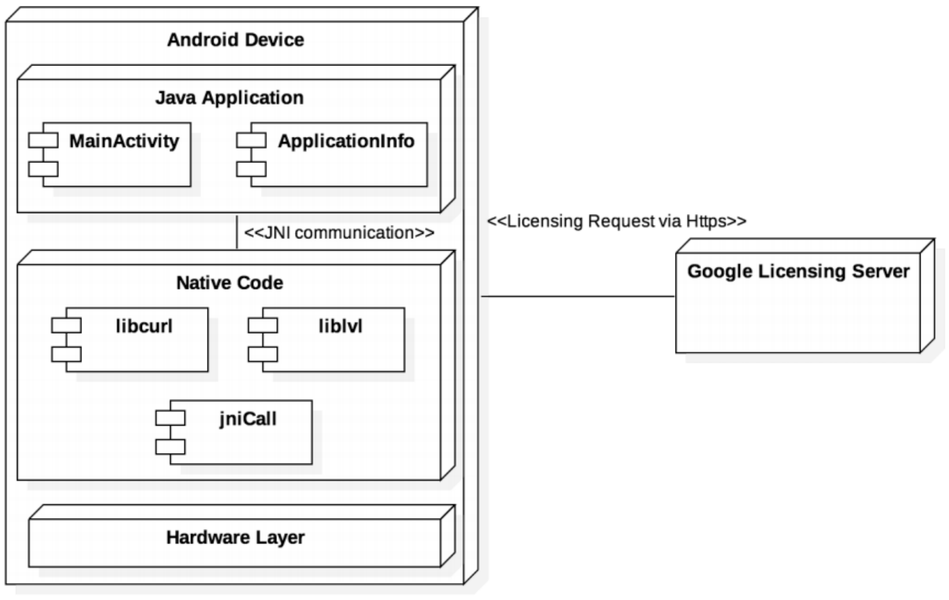


Figure 66 - Overview on the Architectural Design of an application protected by the nLVL [62, p. 55]

Since there was no official support and protocol information by Google available, most information was obtained by reengineering techniques [62, p. 82], which again underlines the insecurity (cf. reengineering possibility) of compiled Java code that is widely used across Google's frameworks as well.

A downside of this approach is that additional permissions and user data now need to be requested by the protected app from Android, since our code is not privileged by Google to be allowed to access this information by default (cf. Google's services have system privileges). Moreover, the user needs to confirm that action upon first start once more. For this reason, we requested that Google considers our solution for official use in the future and invited Google to join a meeting to discuss possible ideas in June 2016 to avoid these additional requirements while looking for options on how to provide the solution to a broad range of developers.

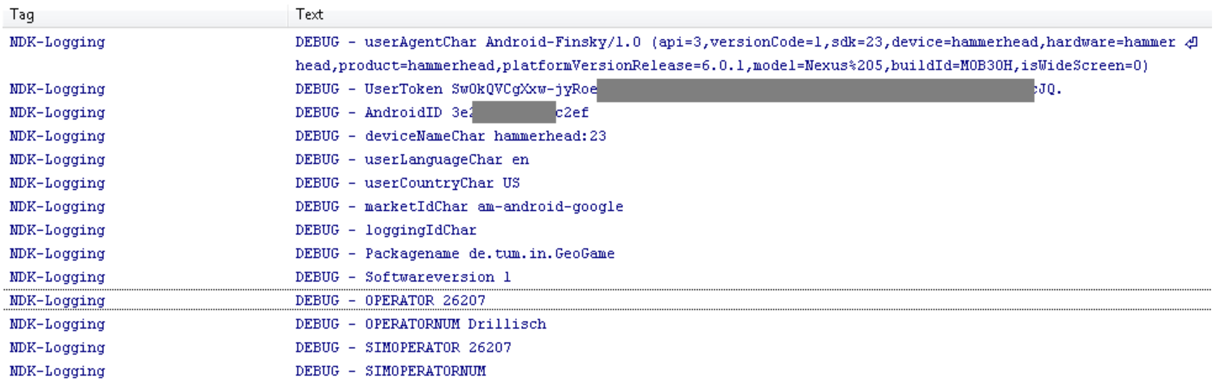
For now, the required permissions are (as indicated in [62, p. 92] and they are now partially minimized in our evaluation efforts):

- android.permission.USE_CREDENTIALS
- android.permission.GET_ACCOUNTS
- android.permission.AUTHENTICATE_ACCOUNTS
- com.google.android.providers.gsf.permission.READ_GSERVICES

However, while the actual information needed to be provided to the native library via JNI [62, p. 90f], some information may even be replaced by fixed values, and did not seem important for the actual license verification and reception of the license response (as elaborated in our evaluation in May 2016, see "fixed" comments in the list below). They are probably used by Google for statistical purposes only. Some devices do not even provide all these attributes (e.g., the market-ID on emulators and loggingID on a Nexus 5 are missing).

According to [62, p. 90f] and the nLVL source codes, this information is as follows, while Figure 67 shows example logging data taken from one of the tests during the development of the evaluation apps. Googles frameworks provide and define this information, while collecting them from different sources (e.g., SIM operatorname from the SIM card).

- User Auth.-Token[134] [unique to user]
- Market-ID [fixed]
- Logging-ID [fixed]
- Device name [fixed]
- User country [fixed]
- User language [fixed]
- Android-ID[135] [unique to device]
- Softwareversion[136] [fixed]
- Operatorname [fixed]
- Operatorname numeric [fixed]
- SIM Operatorname [fixed]
- SIM Operatorname numeric [fixed]
- Packagename [unique to app]
- App Version Code [unique between app versions]
- Nonce [random figure, e.g.1706304994]



```
Tag                     Text
NDK-Logging             DEBUG - userAgentChar Android-Finsky/1.0 (api=3,versionCode=1,sdk=23,device=hammerhead,hardware=hammer ⏎
                        head,product=hammerhead,platformVersionRelease=6.0.1,model=Nexus%205,buildId=M0B30H,isWideScreen=0)
NDK-Logging             DEBUG - UserToken SwOkQVCgXxw-jyRoe[redacted]JQ.
NDK-Logging             DEBUG - AndroidID 3e[redacted]c2ef
NDK-Logging             DEBUG - deviceNameChar hammerhead:23
NDK-Logging             DEBUG - userLanguageChar en
NDK-Logging             DEBUG - userCountryChar US
NDK-Logging             DEBUG - marketIdChar am-android-google
NDK-Logging             DEBUG - loggingIdChar
NDK-Logging             DEBUG - Packagename de.tum.in.GeoGame
NDK-Logging             DEBUG - Softwareversion 1
NDK-Logging             DEBUG - OPERATOR 26207
NDK-Logging             DEBUG - OPERATORNUM Drillisch
NDK-Logging             DEBUG - SIMOPERATOR 26207
NDK-Logging             DEBUG - SIMOPERATORNUM
```

Figure 67 - Example logging data from the nLVL taken from a Nexus 5 device

After acquiring the above information (as originally outlined in [62, p. 58ff]), the properties are placed in a special format that is base64-encoded. This generated string can be send to the Google License server for the actual verification of the provided information. Such a request link may look like the following, shortened example URL call in Table 32 with the string attached in the request parameter.

https://android.clients.google.com/market/api/ApiRequest? version=2&request=CtHY3fJKHFF344k8fdGH […]

Table 32 - URL request to the license servers [based on network.c of nLVL]

As described in [62, p. 65ff] and shown in an overview in Figure 68, in reply, the Google license server returns a zipped licensed information that provides a file “ApiRequest” containing the license response as shown in Table 33. Here the response code is 0 (= licensed), the nonce 1706304994 provided by the developer during the request, the package

[134] Token with limited validity
[135] “64-bit number (as a hex string) […] randomly generated when the user first sets up the device” [175]
[136] of Play Store App

name equals com.appsolution.testnlvl and the version is 2, while that response includes a timestamp as well as an “app-specific user id” [64, p. 55] in addition to a signature, too.

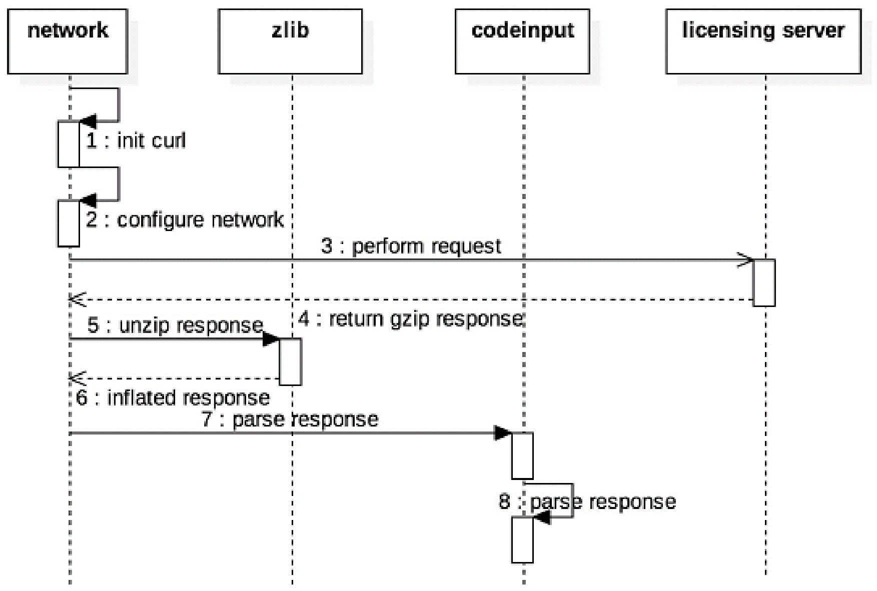


Figure 68 - Sequence of a license request and its response [62, p. 65]

[…]**0**|**1706304994**|**com.appsolution.testnlvl**|**2**|ANlOHQO/GvkUimLYxfoEAPdD43feix23YQ==|14634732706450a/+dAvsn3YqScqnmGcVM […]

Table 33 - Example content of ApiRequest file as supplied by Google License Server in reply to Mr. Chen’s example app

In theory (see known issues below) the next step is to verify the provided data and their signature with the public key as provided by Google in the Developer console of every app. By default, the public key needs to already be embedded in the app project during the development. For additional security, the verification step may be outsourced to a server (see 11.4.9) or to a local secure element (see 11.6.3), before permitting the app access to any server resources, for example. Based on the response code, appropriate measures have to take place (e.g., kill the app upon a negative license reply).

Known issues of the nLVL (proof-of-concept implementation)

The currently implemented proof-of-concept of the nLVL as performed by [62], which is ultimately used in our evaluations as well, still lacks some security issues and the license response’s signature is not currently verified. Moreover, the used library CURL should be shipped with built-in certificates and to allow verification of server certificates. At least one of these features needs to be integrated into a productive version to avoid possible issues with MITM attacks.

Furthermore, the required parameters (see list above) are acquired via JNI and these methods have to remain visible (no ProGuard obfuscation) in order to be accessible from the native code. In theory, attackers are able to modify these functions to provide fake data and use fixed values in a recompiled app to use it across several devices afterwards. Here, Google is in charge of providing more secure native APIs to gain this information without using Java. In general, attackers need to understand the nLVL implementation and its details to circumvent it and many things are not obvious, e.g., within the Java app the methods to provide aforementioned parameters appear to be dead code that is never used. Moreover, our own solutions like indirect method triggering may be used to exchange these information more securely.

11.4.9 Remote attestation to improve LVL

The “remote attestation” [75, p. 85ff] (based on ideas by [322] and [316]) to verify the LVL license status on the server side is another option for improving the license verification and keep access to the servers most assuredly unavailable to pirates.

By default [62] [47], the client app requests a license confirmation from the Google license servers. This is accomplished by collecting various user and device information (see 11.4.8 for further details) and requesting the license server to provide a license response. This response includes the licensing status alongside other data (see 11.4.8 for further details) as well as a signature. This signature and the provided data can be verified using the public key given to every app developer in the Developer Console of the Google Play Store.

Instead of verifying the response by Google’s license servers, only locally (it is the usual implementation and from a security point of view very weak one), the goal is to export the license reply including its signature to verify it externally with the public key. This approach is safe against any modifications (cf. attack on LVL by Xposed / 10.1.5 or a modified Google Play Store from [49]), and any changes will be noticed during the external check to allow appropriate actions, e.g., denying server access. In terms of the nLVL that data is available for further processing by default. However, that approach requires small modifications to the LicenseChecker class (LVL Java version) to obtain the license reply data along with the signature by the Google servers, since it is not forwarded to the developer by default [62] [75, p. 85ff].

Based on the publicly available LVL source codes by Google, a PHP Script to verify the transferred data was created in [75, p. 85ff] and is listed in the Appendix (see 15.1.13). It requires the public key of an application, which may be obtained from the developer console[137], for a specific app and enables developers to verify the submitted data in a secure manner without any possibilities of undetected manipulations. It does this by taking the license response (see Table 34), its signature and verifying it using the public key on the external server [75, p. 86ff].

License response
0|18823373|patrick.lvltest|1|ANlOHQN5Ulh/CIL49nle1l01usO14SSVvQ==|1430363385284”[138]

Signature to license response
Tg1SxIlWAePYAI3j9Pi2 […]

[137] https://play.google.com/apps/publish/

[138] The six values are the actual response code, the nonce, the package name, the “version code of the app”, “an app-specific user id” and the “timestamp included in the request” [64, p.55]

Public Key (cf. Google Play developer console ; only Google owns the private key) MIIBIjANBgkqhkiG9w0BAQE […]

Table 34 – Example for a license response, its signature and a public key [based on [75, p. 85ff]]

11.4.10 Section conclusion

Improving protection in a mostly insecure environment is not an easy task, but as outlined in the recent sections there are several options that can improve the situation, while very few (e.g., SE's generation of a server access token) techniques can even significantly improve the situation for some issues.

Furthermore (see 11.4.1) using more information about the user and the device in combination with copyright protection is reasonable and cannot be faked by others that easily.

The rediscovered methods for dynamic code loading - after Google's removal - provide possible solutions to existing research work that can no longer be used anymore, while generally providing options for hiding code even further. Moreover, the author recommends intensifying the research on providing a method to load shared libraries from memory again, while our current approach allows simply implementations without using any external functions only (see 11.4.3).

The ideas we presented, of fusing Java and native code as shown in 11.4.6 and 11.4.5 that are based on 11.4.4, are of high interest and may be further researched in future research work. We suggest putting any license verifications methods in native code for gaining at least additional security (cf. nLVL / 11.4.8). Moreover, app codes may be ported to native code, while using aforementioned fusing options in addition to prevent the separation.

Ultimately, the usage of different and more secure storage options (e.g., SE) is of interest. Unfortunately with its recent restrictions (cf. 11.4.7), Google limits the possibilities for secure solutions here.

11.5 Further copyright protection options by third parties

The aforementioned sections introduced several options that should prevent users from illegally copying apps, while its protection against static analysis is mainly related to the fact of using native code, since it is much more difficult to understand than the resulting assembly code of DEX files. In addition, we proposed dynamic code loading options that are partly invisible to static analysis tools, e.g., native code obtained from an external source is not visible, while the surrounding methods to call that code can certainly be revealed.

Instead, the presented methods in this section were discovered by other researchers and can be used in combination with the proposals outlined previously.

11.5.1 Preventing static and dynamic analysis

In comparison to the methods introduced in the previous chapter that outlined ideas to increase the difficulty of reengineering attempts, the following two methods try to totally prevent all of the reengineering by stopping it right away. The possible options are highly limited and sometimes are a result of bugs in reengineering tools as in the case of junk bytes.

Junk-Bytes

An option used in the past for preventing the reengineering of DEX files was the use of junk-bytes. As outlined in [[268] as quoted in [64, p. 68]] the method was based on the fact that disassembler by using, e.g., linear-sweep[139] are not able to identify conditional branches like “if (true) jump to address xy”, while (see Figure 69) an instruction following the branch (lines 2 and 3) would have been interpreted by the disassembler and the instructions (here lines 4-6) ignored, and assumed to be the data payload, as shown on the right side of Figure 69.

```
1 if "true" goto line_4
2 load_array_into v1, line_3
3 array_size 10
4 set v2, 1
5 set v3, 1
6 add v1, v2, v4
7 return v4
```

```
1 if "true" goto line_4
2 load_array_into v1, line_3
3 array_size 10
4 0x91 0x12 0x01
5 0x91 0x13 0x01
6 0x92 0x42 0x12 0x25
7 return v4
```

Figure 69 - Explanation to linear sweep issue [64, p. 68]

Obviously this method cannot be applied to modern Android versions using the ART VM, since the code gets compiled and dead code removed upon first installation. Also, it is only a matter of time until the reengineering tools adapt to it, too. Apparently it cannot be used anymore and bugs are resolved in the Dalvik verifier [233].

Emulator discovery

In [224] techniques used by malware were introduced. Since malware often tries to hide its actions, their research results are also of interest for copy protections. Similar to malware, we also want to hide our used methods, too.

For instance, an app may recognize its execution in a simulated environment and instead of using the default methods for license verification it branches directly into the not-licensed case.

The recognition is possible, since emulators, as explained in [224], are using unrealistic sensor values or static values for serial numbers. In fact, during the evaluation phase we found out that recent emulators by Google do not have a defined “ro.serialno” value (see 12.2.3 for details) nor the wifi0 interface, but eth0 instead. This information may be used to detect such an emulated environment.

11.5.2 Methods for protecting Java code

Protecting Java code itself is far more difficult than protecting native code, since many default entry functions (e.g., onCreate(), onStart(), etc.) cannot be obfuscated in any way besides all

[139] Linear-sweep means to execute code line by line without observing conditions

these calls to the Android frameworks. Nevertheless, there are options available that may be used in addition to our proposals.

Obfuscation

One of the default tools to be used is ProGuard [161] and is shipped with Google’s Android SDK. Also, the existing commercial solution DexGuard uses lots of obfuscation methods [272] like arithmetic obfuscation, control flow obfuscation, name obfuscation or resource obfuscation aside from many others.

Furthermore, manual obfuscation (= renaming of functions/variables, concatenating strings with several variables, adding nonsense [158] code like calculations in-between etc.) may be an option in cases where the usual automatic obfuscation cannot be used. For example, native code requires to know function names (hardcoded) that will be called from native code (cf. nLVL and required user/device details) and ProGuard needs configuration in a way that exempts files from obfuscation. For instance, this applies to Amazon’s DRM as well and using manual obfuscation may have increased the difficulty for reengineering, since the Boolean variable “drmenabled” in the kiwi class (= main class of Amazon’s DRM) revealed its function with its name completely and allowed to disable the protection within seconds [241].

Encryption

A typical way to hide information is to encrypt data, while using obfuscation and complex algorithms to make decryption for third parties that do not know the details, more difficult. For instance, in [236] researchers used encrypted program code, while dynamically executing it. Besides the previously mentioned limitation (see 11.4.3 regarding dynamic code loading), that approach is still possible today. Moreover, the commercial tool DexGuard provides several encryption methods [272] like class-, string- and WebView encryption.

The “DIVILAR” [162] solution instead is using different opcodes as a reengineering protection. That approach is not possible with modern Android versions anymore, and it would not get compiled by Android >5.x (using ART VM) upon installation. It can still be of interest as a special encryption variation for dynamically loaded code that gets modified on-the-fly (= decrypted) to load it afterwards (see 11.4.3 for details on dynamic code loading).

Protection against dynamic analysis / attacks

As one of the more advanced tools, DexGuard has to again be mentioned for providing many methods against reengineering, e.g., certificate checks, emulator detection, debug detection, root detection, tamper detection, and SSL pinning that allows developers to take actions against these attacks [272].

Countermeasures against hooking of functions

The hooking of functions is a threat to both Java and native functions. Most developers often are not aware of that issue, nor is it possible for them to detect a manipulation at runtime (cf. LVL hack in 10.1.5) without specialized countermeasures.

In [298], several options are mentioned that allow detection and acting on it by:

1) using the Package Manager to look for suspicious applications (e.g., de.robv.android.xposed.installer or com.saurik.substrate)

2) monitoring app’s stack traces for abnormal calls (e.g., de.robv.android.xposed.XposedBridge->handleHookedMethod) to detect active hooking
3) checking the memory mappings /proc/[pid]/maps for libraries by these hooking frameworks (cf. pathname and values like /data/data/de.robv.android.xposed.installer/bin/XposedBridge.jar or /data/app-lib/com.saurik.substrate-1/libsubstrate.so) to detect active hooking as well

In case of a detected hooking, an application can perform several tasks, such as notifying the developers of such an attack. This can be done by providing the IP address and other information to take legal actions, since hacking an application is considered a crime in many countries now (e.g., Germany: §95a UrhG, which describes that it is not permitted to circumvent a protection [323]). If such a notification is legal in this case needs to be discussed with a data privacy officer and was not further reviewed, since legal questions are out of scope of this dissertation.

11.5.3 Methods for protection native Code

Static functions

A heavily missed feature in Java language to hide functions is the “static” directive. This is available in C to restrict functions to local usage that will not get exported or be available for linking. An example taken from [62, p. 75f] is shown in the Appendix (see 15.1.14). By using the tool nm[140] on Linux, it is possible to take a look at the exportable functions of a library and those using static will not be mentioned anymore. In theory, (using outdated “Cydia Substrate” [152]) and without obfuscation involved, it is possible to recover these hidden functions by knowing some internal information. This information can be used methods or return values to locate the address of the target function within a native library that may be used at runtime, and in combination with the (upon the runtime available) base address of that library to make it callable again [281].

Strip

The strip command, as explained in [62, p. 77], can be used to remove any debug information as well as the symbol table from an executable file. This method can also be added to the compiler flags with parameter “-s” to be used by the Android NDK. An example taken from [62, p. 77] showing the differences on assembly level can be viewed in the Appendix (see 15.1.15). Due to dynamic loading of libraries, the addresses that any symbols point to, are specified during runtime anyway while some symbols names will remain in the .dynsym section.

Pragmas, visibility-attribute and -flag

Hiding symbols GCC 4.x, as explained in [324] and [325], offers two additional options using either pragma statements to hide several symbols (function names), or by using the visibility attribute as shown in the source code samples the Appendix (see 15.1.16). Furthermore, there

[140] e.g. nm -aDC --defined-only example-arm.lib [281]

is a compiler flag "-fvisibility=hidden" that may also be applied. Asides from the security benefit, not exporting all symbols can significantly decrease the loading times of the shared libraries.

Hiding data in binaries

In thinking about hiding decryption keys or encrypted data, GCC provides an interesting option to hide information, as discovered by [326] by using the naked attribute. By default, gcc adds several operations to a defined function like saving "the current frame pointer and load[ing] the function arguments into the appropriate registers […] [, restoring] the original stack […] and eventually […] jump[ing] back to the previous address" [326]. These extra operations can be avoided by using the aforementioned attribute "naked" as shown in the sample source codes in the Appendix (see 15.1.17). According to [326], it results in confusion of all major disassemblers like objdump, Hopper, or IDA that are trying to recognize opcodes with their parameters instead of data.

Hardware Dongles

Pure native code is best protected when it is inaccessible to attackers. The introduced SEs allow embedding of Java code applets only[141]. A similar solution called "Guardant Code" [230] allows the executing of native code in an external device, while returning the execution results. Nevertheless, their currently provided framework lacks the same issues as any java-based framework used by many SE vendors and may be intercepted quite easily (cf. Xposed Framework). By assuming it is used for performant decryption of data, it could still be an interesting solution and any decryption keys can remain safely stored in the dongle.

Protection Software

In the most recent version, DexGuard provides native code obfuscation and -encryption [272]. Moreover, the introduced Obfuscator-LLVM (see 10.3.3) can be used to obfuscate the native code.

11.5.4 Identification of pirated apps

Besides actual copyright protection, a subtopic identifies those that distributed an app illegally, while it is currently not possible to identify the source when an app is provided on a warez website[142]. Right now and as mentioned before, APK files on each system are still equal when bought on the Google Play Store. As a solution, researchers propose in [234] using watermarks specific to a user to be integrated into the DEX file by "reordering the sequence of instructions" [234]. This idea is similar to our idea of including device- and user specific attributes for copyright protection. Both ways may be used for the proposed identification and ultimately to take legal actions against those distributing apps.

[141] by default. The flash controller can be programmed in C by G&D only.
[142] Website offering illegal goods such as pirated software etc.

11.6 Open possibilities for using SE and native code

As previously mentioned, using the native libaums library for improved security access and interaction with SEs is not possible, the following ideas cannot be realized right now either and require modifications to stock Android by either Google or a device manufacturer that permit the native code to access USB devices again. The detailed reasons and proofs are in 11.4.7.

Assuming that these issues are solved, the following pages present several solution ideas for using SEs in combination with Android Apps for improved copyright protection. As stated in the fundamental section, a secure element provides a secure space by definition [327]. While some logic may be executed internally (e.g., calculate checksum of 1KB of data), other performance-intensive tasks (e.g., decryption of 1GB data files) need to be outsourced to Android and preferably in a secure manner (= usage of native code and obfuscation options to protect keys in the insecure Android environment).

During the early days of this research work, our initial approaches still targeted Android on the Java level, and we were not aware of the severe security issues of Android. These approaches included the following:

- library “libaums”[143] [199] for accessing a via a microUSB adapter[144] connected SDcard like the MSC by Giesecke & Devrient (embedded SE)
- a Java-based proxy application that allows a secure information exchange between a SE with JavaCard 2.x (usually just a slave device) and a server (cf. license information exchange) as implemented in [197]
- a first attempt at creating a simple copyright protection using both previous works as outlined in [328] and shown in the form of a sequence diagram in Figure 70. Here, the MSC represents the SE that is used for storing the decryption keys to protect the app content. This application was used in the first evaluation approach.

Nevertheless, these solutions were abandoned when discovering the confirmation for previous assumptions (see 13.3) that using Java code cannot be the best approach at the current time[145].

For that reason and with that early assumption in mind, a native version of the libaums library was created and implemented in [201]. Even Google does not recommend using the native code stating it “has little value for many types of Android apps” [329], which is obviously not true for security-related purposes. Moreover, the development process of the native libaums allowed an early insight into the issues with using native code only, since the Android NDK does not support accessing USB devices. The workaround was to acquire the file identifier via Java (cf. required Android permission) and hand over this information to a modified version of libUSB in the native C source code. Unfortunately, that solution does not seem to work anymore on newer Android versions, as outlined earlier. An overview on the implementation is shown in Figure 72, while providing an overview on the original Java version of that library in Figure

[143] https://github.com/mjdev/libaums
[144] e.g. http://www.meenova.com/st/p/m3r.html#devList
[145] Notice: This might change when Google decides to use native code within ART VM only, and it is the recommended solution for both app developers and Google to use native code as much as possible. Further information on this are presented in the native code related sections of this chapter

71. It needs to be noted that the libUSB used by the Android frameworks has the required privileges and therefore the Java version still works.

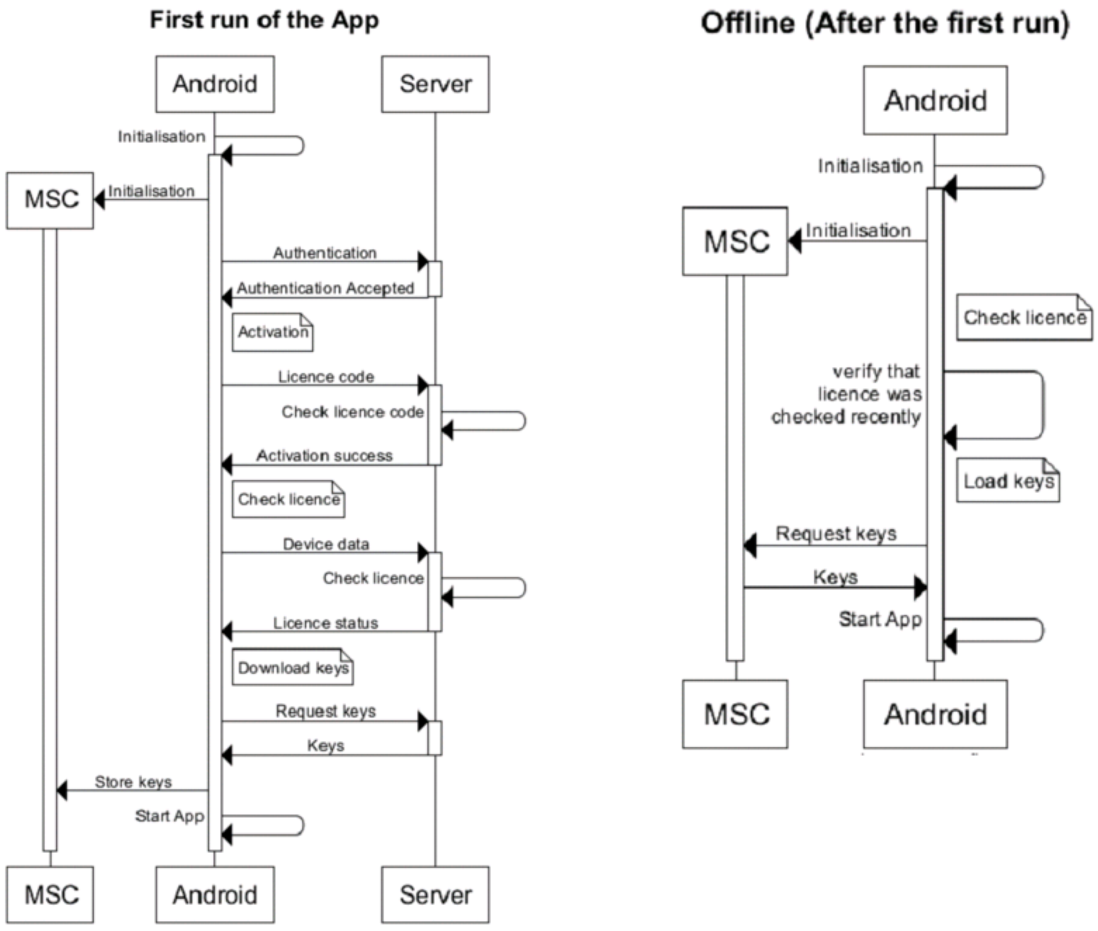


Figure 70 - Sequence diagram of our initial copyright protection approach using SE and Java only (based on [330])

By now having access to a native libaums, it is possible (cf. issue / target Android version < Android L only) to use the secure element in combination with native C source code to store and exchange encryption keys, license information or trigger small applets within the secure element. This allows us to execute various tasks and everything in a much more secure manner other than by using the original libaums Java version.

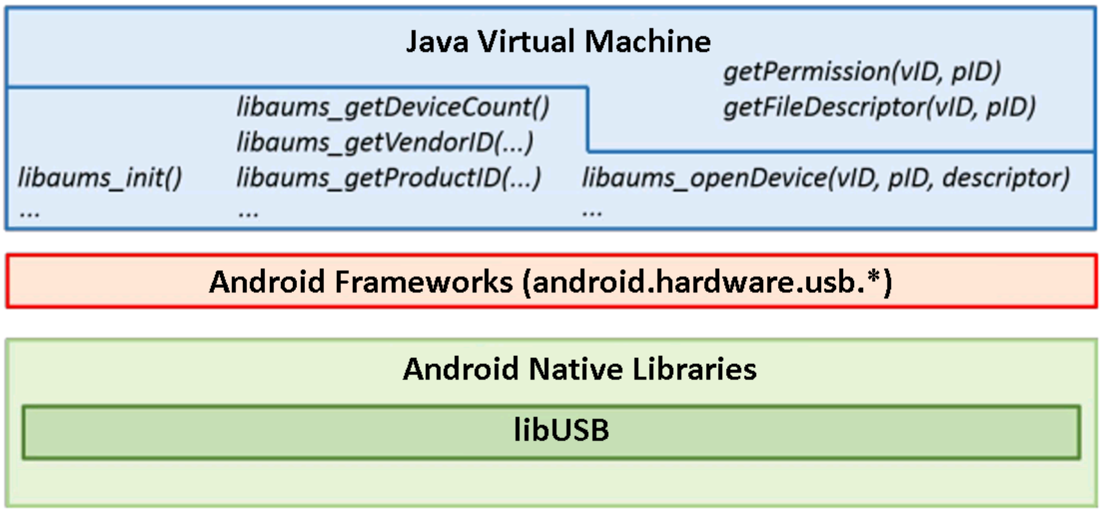


Figure 71 - Libaums implementation (Java code only) [based on [201] and [199]]

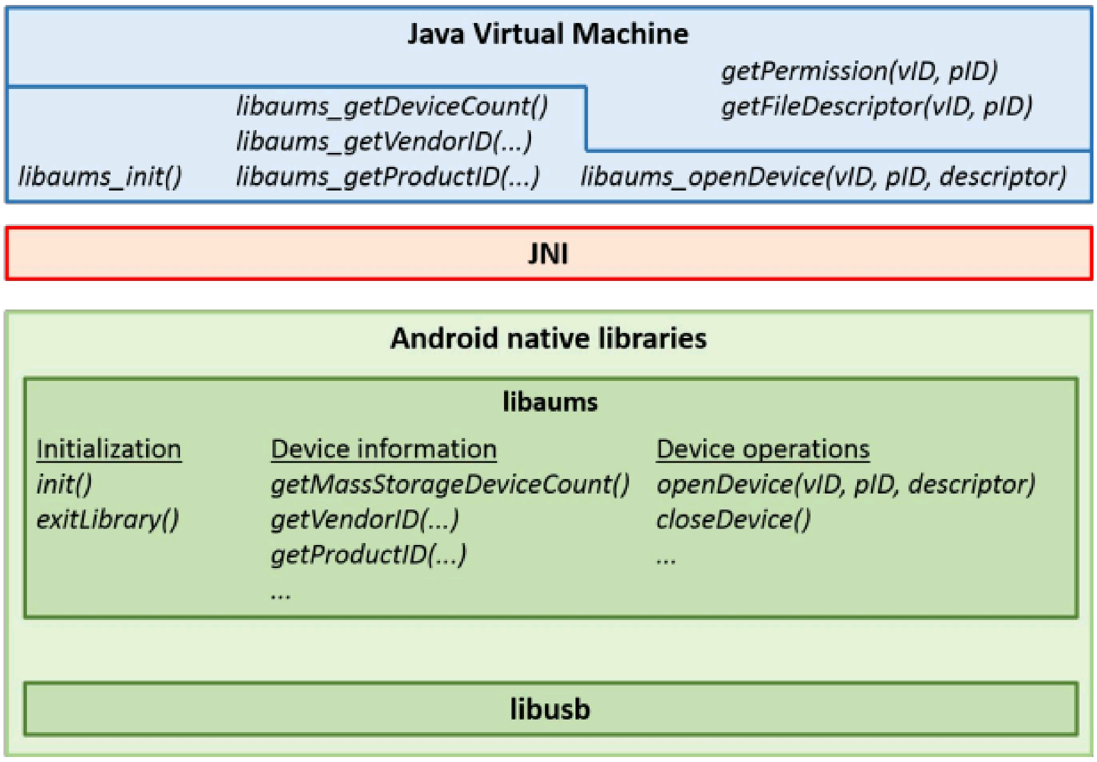


Figure 72 - Overview on the native libaums implementation (native code with minor Java code parts) [201]

The following sections present conceptual examples for using secure elements, in combination with Android for improved copyright protection.

11.6.1 Secure local storage

As mentioned by GlobalPlatform [327], an SE provides a secure space for confidential information, like in the current case, licensing information.

For instance, the SE may provide access to stored data by entering a PIN only (typical example applet provided by G&D). While the used PIN would be very vulnerable in Java source code, it can be highly protected in native code. Moreover, the information exchange between a SE and the native code is more secure due to more difficult reengineering, too.

In terms of copy protection, the SE may be used to mirror licensing data from an external server providing customers licensing options even without an internet connection, e.g., on a flight or a long journey. See 11.6.2 for an example.

This approach may be considered as safe as using native code and file encryption, but even here a SE may be used to store the used confidential data more securely, because presumably many developers (attackers) are not used to secure elements and using them instead is an additional barrier.

11.6.2 Secure local license provider

A license provider or a license server is a typical server that organizes the licensing of apps, stores customer data and has information about the payments. A simplified database on such a server to organize licensing may look like the one shown in Table 35. It consists of a unique SE id as well as an app id and related user- and device data for identification purposes, while storing the license status. For instance, the entries could be created upon the first run of the application, while the license status can be verified externally depending on the app store and its provided interfaces for payment verification.

Unique SE ID	**Unique App ID**	**UserData**	**DeviceData**	**LicenseStatus**

Table 35 - Possible database structure of a licensing server in combination with using SE and device/user identification

Here, a SE may be used to cache the licensing status in a secure manner, while acting as a middle layer between the client (Android application) and a server (licensing management). A great advantage is the possibility to provide many functionalities offline after an initial information synchronization.

For instance, user and device identification information (cf. 11.4.1) may be transferred and stored on the SE. It can be used to bind it to the smart device by checking these information upon further requests (e.g., each time a native function is used), while several information (e.g., decryption keys) may be provided dynamically only, now. That makes the secure element mandatory for the app.

In addition, the SE may transmit and validate the device and user information alongside a unique app and unique SE ID to the licensing server (via aforementioned proxy app, but here in a native version) to either initially register or later verify the licensing status based on the given information.

The licensing server may keep track of the actual licensing status by verifying any payments externally (e.g., by checking of banking account for transaction) on the server that cannot be modified by the app attackers.

Right now, the available copyright protections like the LVL have the issue that license checks as well as any signature checks take place on the insecure device locally, and therefore, in an insecure way (Java), too.

Instead, app developers are requested to perform these checks in a secure manner at least (e.g., by using nLVL / 11.4.8) or to perform these checks even externally (see 11.4.9) for best security protection, while providing access to server resources (e.g., game data or server connection for multiplayer games) upon successful verification only.

11.6.3 Verifying the LVL signature within the secure element

Similar to the method introduce in 11.4.9, the secure element can be used as a local verification option instead of outsourcing the check to an external server and updating the internal license status accordingly. The information can be used to modify the future behavior of the secure element. For instance, a negative check may result in blocking future requests by the device/app, which naturally results in a non-working app.

11.6.4 Secure server access

While the secure element can hold the license status in addition to a license server in general, it can also be used to limit access to server resources in a secure manner by making it the mandatory tool for server access. This idea was outlined previously in 11.4.7.

Here, the server supplies a signed timestamp to add any time restrictions, since the SE does not own an RTC and Android cannot be trusted. Using a shared secret (with the server) the SE can verify the timestamp’s signature and once more sign the whole timestamp with its signature to allow the server the verification and any time limitations for server requests. While using a public-private-key method for the signing an even simpler approach can be to use a shared secret on the server and SE that gets added to the timestamp and hashed by a “One-way [secure] hash function” [331] like md5 as already explained in 11.4.7. While the shared secret cannot be revealed that way, the server as well as SE can verify and reproduce the hashed values for verification purposes.

In theory, this approach is not completely safe against so called “cardsharing” [292] (see more details in 10.4.1). This is difficult to address, since many SEs do not provide an internal RTC to detect rapid requests for more than one client app that is distributing the tokens to other smartphones. Nevertheless, due to aforementioned implementation the server may be used to fix that issue here, while now permitting requests within a certain time frame (cf. solutions for cardsharing [293]). For instance, as illustrated in Table 36, a typical request takes 63.5ms latency using 3G on average [332], and while accessing an SE using libaums library with up to 255 bytes (sufficient for a timestamp with two MD5 hash values) as the parameters (see 8.5.1) that takes, on average, about 200ms. Therefore, the server should expect the clients request within ca. 350ms, while illegal accesses by other smartphones (cf. forwarded access information) can be expected about 65.5ms later. Of course, that is a very theoretical calculation and delays might be caused by a busy CPU on the server as well as client side or network failures that require resending of information. Moreover, Android itself is not a real-time OS, too. Ultimately, there are lots of sources for delays. Nevertheless it is an approach to limit issues related to cardsharing [293] and manufacturers are advised to integrate a clock feature into the products to detect rapid access and assumed misuse of SEs.

App requests sign. timestamp from server via 3g	(1) 63.5ms [332] + ~1ms calc. time (assumed)
App contacts SE via libaums with signed timest.	(2) 200ms (including processing time)
App contacts server with sign. signed timestamp	(3) 63.5ms [332] + ~1ms calc. time (assumed)
	= 329ms → ca. 350ms recommended
Attacker app forwards it to other clients over 3g	(3) 63.5ms [332] + ~1ms calc. time (assumed)
Other smartphone accesses server	(4) 63.5ms [332] + ~1ms calc. time (assumed)
	= 394,5ms (expected arrival time / 3rd party)

Table 36 - Required time till server access in comparison to 3rd party clients

11.6.5 Outsourcing program logic

Due to the aforementioned performance issues, the following approach is currently highly limited for an actual implementation, and the performance of a SE is too weak, besides missing typical Java frameworks within the SE's Java environment (see 8.5.1 for details).

While it can be categorized as kind of a fusing option as well, by binding an SE and app together, the idea is to outsource parts of that app to an SE to be called from the app for result values. This approach might be used for calculation function requiring some input values, while perhaps returning the computed results. Therefore, it depends on the app in such a way that it is suitable for a specific app. The general idea is not new, and companies like Aktiv Soft JSC provide such a solution with their product of an external dongle that already uses native code [230].

In light of the upcoming Project ARA[146] by Google, with shipments of early developer editions expected this fall (2016) [333], the idea becomes even more realistic providing developers the opportunity to develop their very own, specific, high-performance modules for such a purpose. This will also allow companies to develop additional, more generic solutions for a module to execute outsourced code from different vendors, previously installed through a secure channel

[146] Project ARA is a modular phone consisting of lots of exchangeable modules to adjust the hardware [333]

and administrated by that company. The requirement for a physical, secured module would limit piracy instantly due to the fact that the outsourced code is transmitted securely and never available to customers except in form of a module that is yet to be developed.

11.7 Overview and best solution approach (example)

A question that is often raised is, is there a very best solution available? It is not easy to reply to this question, but one certainly needs to quote a statement by Thomas Aura et al. that is still valid after a decade - “Copy protection is never perfect” [1]:

It highly depends on the skills of the attackers, besides the chosen options for protecting apps by the developers.

In this section, the best approaches are once again highlighted by taking a more generalized view on the issue, before presenting an implementation idea with regard to the proposed solutions of chapter 11.

11.7.1 Best approaches in general

- It is highly recommended to use native code, which is much better protected from reengineering (see 10.3ff, 11.1.2 and 11.5.3).
- Therefore, implementing the nLVL (see 11.4.8), along with fusing options/indirect method triggering (see 11.4.4, 11.4.5 11.4.6) is the suggested way. If parts of the main app can be transformed to native code, then it is highly recommended to bind native and Java code even more strongly together; thereby making them mandatory to each other.
- For additional security and in terms of huge applications, dynamic code loading (see 11.4.3) is suggested to load missing program parts after the license was verified using, e.g., the nLVL. In general, our shown method of loading assembly code from memory can be used to modify app parts on-the-fly, e.g., update embedded encryption keys to correct ones.
- If the chosen license requires limiting the usage to a user or a device the desired identification sources (see 11.4.1) need to be selected and combined with the license check. Here the preferred options are those ones that can be realized using native code only. If that is not possible due to the chosen license option, the realization using JNI and calls from native code are recommended. The information itself needs to be stored externally, e.g., on a secure element or an external server along with an identifier (e.g., the Google account) to reapply the license after a device reset.
- SEs should be used to provide access keys to servers (see 11.6.4 and 11.4.7)
- Existing obfuscation tools and other methods protecting source codes have to be applied (see 11.5)

11.7.2 Best solution approach by example

The following best solution approach is an example only. The selected solutions depend on the defined license. For instance, it is not required to integrate any user identifications, when the target license is to run an app per device only. Instead if the license is defined to allow the usage across devices and for a specific user only, then attributes about the user need to be integrated.

1) Define the license

First, the desired license (see 11.3) needs to be clarified by a developer. The recommended license might be the aforementioned “One Device, One User/Many Users” license, since, e.g., family members will share their account on a tablet device anyway while the vendor wants to resell the app, if it is simultaneously used on two devices.

In that scenario the device needs to be identified in addition to also verifying the actual payment for the app by using Google’s LVL services or an external provider. This depends on the origin of the app, where the user bought it initially. In this example we will think of the Play Store by Google.

2) Select the preferred methods to verify the license

While Google’s license of one user account is enforced by using an app already purchased on Google’s Play Store, the customized protection may now add an option for the device’s identification. In general, all solutions should be implemented in native code (see evaluation results in 13.4 and the results of previous chapters). Therefore, the nLVL needs to acquire the license status of that app using the Google account in general, and then indirectly receives the confirmation for a valid payment hereby (if the license is valid, the app was purchased).

As outlined in 11.4.1, the best options for identifying a device are those that can be used within native code, and here preferably those ones that cannot be modified that easily. If the vendor prefers a reliable identification, he needs to choose those that offer a 100% identification rate like unique numbers, e.g., the serial number of a device or the hardware address of the used wireless device. Both pieces of information can be fetched from system files or system properties.

Using more than one method is recommended, and every additionally used option may block an attacker from cracking that app. On the other hand, the performance of an app might suffer from too many used methods, and it is ultimately a questions of performance tests by the developer to include as many options as possible, while keeping the performance on an acceptable level. While methods gathering static information return almost instantly, more complex identification methods require several seconds and others even hours or days (see 11.4.1 for all these details).

Since a SE cannot be used at the moment to store information permanently, the Google user account needs to be stored by the developer along with the device attributes, app package name and app version on, e.g., a license server.

In case of the user installing the app on another device, the developer’s license server can indicate towards the copy protection now that the app was installed on a different device that now triggers the desired actions by the copy protection. Moreover, after a device reset and new installation of the app the valid license can be restored.

3) Select the options to protect and enforce the license

While the actual license check could have already been done with existing solutions (e.g., default LVL) in an insecure way, the benefit of using native code for device identification, asides license verification, is its better protection against reengineering (cf. 13.4 for evaluation results). Since most codes of an app are developed in Java, another core issue is to keep native code and Java together by trying to fight their separation. For instance, attackers may try to get rid of the native library by hooking it (see 10.3.4).

Therefore, methods for “fusing code” (see 11.4.6) should be integrated into the code as much as possible as long as it does not affect the performance and can be reasonably hidden, which depends on the actual app size. Here the reason is that Java code remains highly insecure in terms of reengineering (see 10.1ff), and it has to be assumed that attackers will identify the one or another method (fusing option), especially if it looks suspicious such as triggering an exit command (cf. evaluation / see 13.4). One of these methods might be the character array comparison as introduced in 11.4.4 that triggers an action if the native code does not modify the condition upon positive license reply.

In our evaluations we noticed that (simulated) attackers try to look for functions quitting the app. Therefore, using options such as “Indirect Method triggering” (as indicated in 11.4.5) can be used to trigger desired actions later, e.g., to render a game useless by modify game values during the game that have a very negative impact on the gameplay, or by causing the app to quit after a minute maybe. While this should be hidden as well as possible, e.g., by executing a dynamically created shell script that quits the app or by loading dynamic code that triggers the kill command or by accessing a wrong memory address resulting in a segmentation fault error. The goal is that an attacker should have issues to understand how this function was triggered and even if it was triggered at all. Of course, this results in a bad user experience and possibly poor reviews by software pirates. It is up to the developer to select the desired actions in case that a pirated app was identified.

4) Apply further protection options to harden it against reengineering

In addition to the mentioned proposals all existing protection mechanisms may be used, e.g. ProGuard for obfuscation or its improved, commercial version DexGuard (see [161] [160]). Here we note that Java code obfuscation and native code are prone to some issues and functions used via JNI should not be obfuscated, since they are usually hard-coded in the native libraries (e.g., the methods to get the user’s token in our nLVL implementation). A possible solution could be the manual obfuscation and instead of using a meaningful name, codenames may be used, while adding the actual name as a comment in the source code for the developer himself. Morever, indirect method triggering can be used to exchange information in a more secure way, e.g., by placing these parameters in environment variables instead.

Furthermore, Java code obfuscation - particularly on Android - is less effective than native code obfuscation, e.g., function signatures remain intact (even using obfuscated names), while C source code allows it to hide them instead (see 11.5.3). Moreover, many external library calls in Java code remain the same, e.g., methods like onCreate() and onPause(). Furthermore, the ProGuard obfuscation revealed itself as not a real barrier in the performed evaluations and students (here: attacker role) simply looked for methods, such as exit calls, to identify the interesting, obfuscated classes (for details see evaluation results in 13.3 and 13.4).

In addition, obfuscating native code is heavily researched by others and besides obfuscation options like Obfuscator-LLVM (see 10.3.3) even usual compilers offer interesting flags and directives (see 11.5.3) to hide information, like encryption keys using the naked attribute [326].

In general, app data can be shipped encrypted as an additional barrier, while decrypting it using the aforementioned, hidden keys on-the-fly. Thinking about SQLite, there are even libraries like sqlcipher[147] providing that functionality. Nevertheless, it needs to be highlighted that decryption procedures should take place in native code and strongly linked to the license status.

All these methods buy time only, but ultimately cannot prevent the cracking.

[147] https://www.zetetic.net/sqlcipher/sqlcipher-for-android/

12 Prototypic implementation

Based on some of the ideas in chapter 11, developers want to select their desired functions and develop their very own unique protection solutions to prevent app piracy from happening within a desired time frame, while observing the target license (see 11.3).

The actual time an attacker is required to circumvent a protection cannot be measured and depends highly on the skills of an attacker and ultimately, on those of the developers trying to hide certain methods.

While every additional used method may increase the security level, some of them may affect the performance of an application. In the end, the author recommends following the best practices (see 11.1 and 11.7) for improved protection.

The following chapter illustrates the protection of two apps that were protected differently. One is a Java-only implementation, while the other one comes in two differently protected versions that both are partly implemented in native code. They are explained in detail below and have been used for the evaluation.

12.1 Demo applications

For testing our copyright protection mechanisms we used two different, existing programs to apply the protections. The first one was an open-source game called ReGeX [334], as chosen by the student implementing the protection based on the ideas of using a secure element and content encryption. The author of this dissertation chose the second program as a result of our first evaluation. Here, it turned out that knowing the source code (ReGeX is an open-source game) allowed attackers to gain too much insight, in a very short amount of time to identify certain protections more easily. Therefore, the 2nd implementation makes use of the closed-source game SignPost, and was divided into a more java-based version (like in the 1st evaluation, but some native code ideas) and a native-based version using native libraries and several fusing options for protection heavily. The implementation sections below outline the details.

12.1.1 ReGeX

ReGeX [334] is a open-source game that asks the players to find regular expressions that “match [a] certain string, but doesn’t match others” [334].

12.1.2 SignPosts

Mr. Tim Falkenmayer, Mrs. Elisabeth Braendle and Mr. Alexander Ostrovsky, who kindly permitted the usage for our research purposes, developed the game “SignPosts” [335] in a former Android Practical Course in 2012. As mentioned already, a significant reason for choosing this app was the fact that is was closed-source, and also quite complex in order to allow the hiding of certain protections within the source code.

The game consists of coins with city names that are dragged and dropped at the right angle and distance between two defined cities to acquire points or lose lives as shown in Figure 73 [335]. The game uses "AndEngine" [336], which is a free and open-source 2D OpenGL Game Engine.

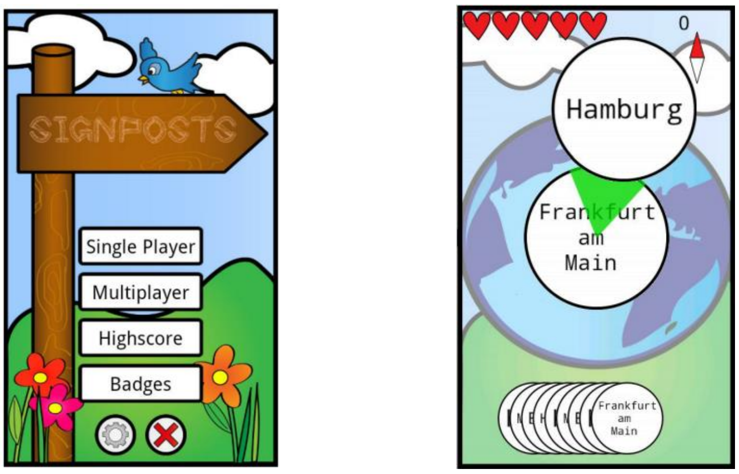


Figure 73 – SignPosts App (not published by authors) [335]

Figure 74 shows in a state chart the typical flow of a game for earning points by matching a city, e.g., Hamburg in a correct angle to Frankfurt, and then a player guesses the correct distance. If that guess was not good enough, the player will loss a life indicated by the hearts.

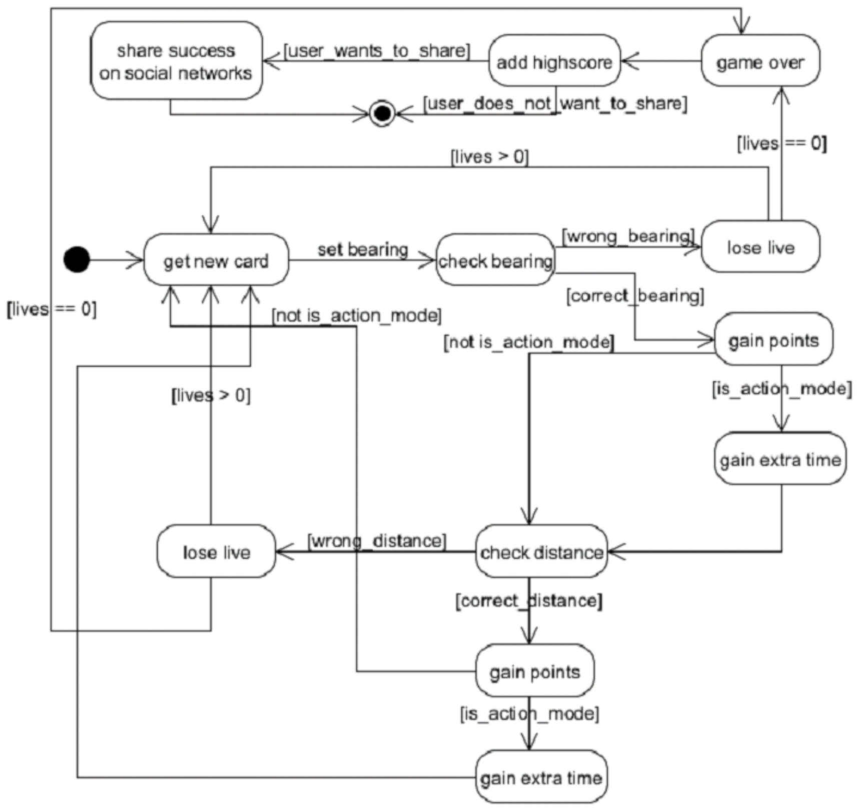


Figure 74 - State chart describing the game (classic- and action mode) [337]

12.2 Actual implementation/injection of the protection

The current section focuses on the implementations of the Copyright Protection (CP) methods injected into the existing apps, but it does not explain the implementations of the actual games. This information can be obtained from the website or project-specific documentations instead (see [337] and [334]).

12.2.1 CP implementations for eval. 1 using SEs and ReGeX

This app protection was implemented by [328], while the general implementation plan and usage of previous works (cf. [197], [338]) was discussed in advance and in compliance with the proposed ideas (cf. [186]). This solution consists of four major parts – the app with services itself, the licensing server, the SE, and a desktop tool for developers. These four entities are presented next in more detail.

App

Similar to real developers, the student implementing the protection on request was asked to choose the preferred information sources himself (see 11.4.1 or [302]). The following information was chosen and obtained by the protection outlined in [328, p. 11]:

- Android ID
- GSF ID
- Serial number
- IMEI
- Information about the device like manufacturer, model and CPU
- MAC address of the Bluetooth hardware
- MAC address of the Wireless hardware

The protection works in the way that this information [328, p. 11] is send to the server (using an SE, see next sections) upon initial activation and on each app's start. Furthermore, the protection used the default ProGuard obfuscation in addition to further content encryption using a key that was obtained from the server or later (after initial receiving) from the SE [328, p. 12]. Initially, the encrypted content was created by editing the app's source code with a special tool called either "PC" [338] (initial version in a previous thesis) or "PC-Tool" [330] that exchanged strings with encrypted ones and functions that call for decryption (see next sections for details).

The used license, in these cases, was to bind the app to a single device/SE, and a one-time license key was used for the initial registration of an app and its device/SE with the licensing server [328, p. 12].

Licensing Server

The implemented server was mainly designed to organize the license verification, while providing required keys used by the app for decryption purposes [328, p. 12].

In detail, the licensing server provides the following functionality as outlined in [328, p. 13f]:

- **Authentication**: The SE connects to the server using a proxy service shipped with the Android App including the access drivers for the SE provided by [197]. While this connection is already encrypted it uses an internal SE ID for identification purposes.

- **App Activation**: Upon initial registration, the app is activated by supplying a predefined license code (valid one time only), the name/version of the app, the SE ID as well as device-specific information (see listing above). If the license code is accepted, the server's database is updated with the mentioned information.

- **License Validation**: From time to time the license status is verified and aforementioned information (App name and version, SE ID, device information) sent to the server, while receiving the appropriate response, if the SE ID owns a valid license.

- **Key download**: One of the key features in this approach is the content encryption. By providing the app name and version as well as the SE ID the server verifies the authentication of the SE and if it holds a valid license while providing the required decryption keys in reply.

- **Store device information**: This function provides the ability that allows clients to upload device information to the server.

- **Register new apps:** The username/password protected function is only provided to developers and in conjunction with the so-called PC tool as explained next. This allows the registration of an app with the used keys that the PC tool uses in regard to the app encryption.

Further details on the detailed implementation and communication using PHP and Apache server are available in the thesis itself (see [328, p. 17ff]). Even more details on the access on SEs that are based on earlier implementation by the author himself (which are based on official driver source codes by G&D itself) and improved in another thesis can be found in [197].

PC-Tool

The "PC-Tool" [328, p. 20f] is a desktop tool designed for use by developers for content encryption of an app before its compilation. It also allows the initial app registration to provide the license server the encryption key for a specific app version. Similar as performed by other tools (e.g., DexGuard [160]), strings were selected for the encryption, by replacing them with function calls with embedded and encrypted data that ultimately need the decryption key by the server or a secure element upon runtime.

Applet on the SE

An additional component used in the implementation as outlined in [328, p. 20] is the SE with its applet. Here, the introduced MSC by Giesecke & Devrient (see 8.5.1) was used. The applet is compiled with a unique ID (cf. SE ID) for identification purposes. The SE is used to store the decryption key after the successful authentication and key download as explained previously. The access is protected by using a special code besides the general special SE access that needs to be embedded into the Android app. This is an insecurity that cannot be avoided and which led to our additional ideas of using native code (see next implementation). In general, the key exchange with Android is required, because of the weak performance of the SE (see performance section of 8.5.1).

12.2.2 CP implementations for eval. 2 using the nLVL/minor fusing and SignPost

Based on the results of our evaluation in January 2016 (see 13.3), a closed-source application (game) called "SignPost" [335] was selected for applying copy protection methods, while preferably using native-code solutions this time. Due to the strong limitations of using secure elements (see 11.4.7) any related methods were not evaluated here and remain conceptual ideas.

nLVL

In an approach to secure the license verification by Google known as LVL [47] the idea of a native implementation was first introduced by the author at the Android Security Symposium in Vienna [164] and implemented, on request in a thesis shortly thereafter [62]. The implementation details of the nLVL were previously introduced in 11.4.8. The library was added (in binary) to the game by calling it via the JNI attached native code. Therefore, the only function visible and used in Java is the native function, getLicenseStatus().

Upon the creation of the protected game by the author of this dissertation, it was observed that many previously fetched (see 11.4.8) parameters (that sometimes were not available and resulted in native crashes in the original authors' thesis implementation) were not required for the actual license verification and here replaced by dummy data. Ultimately, it remains unclear whether Google uses these other information for statistical reasons only, or if it is somehow used for licensing, while providing fixed values works perfectly. In sum, the following information is still acquired by calling the respective Java functions from native code via JNI[148]:

- user's AuthToken
- AndroidID
- Package name
- Software version

Furthermore, open issues were detected that would allow MITM attacks in the current proof-of-concept implementation by [62] and the LVL was not completely ported by the original author. In particular, the verification of the signature of the supplied data by the Google servers is missing (e.g., by using OpenSSL) and the used CURL library is not configured to verify a server's certificate. For productive usage one of these issues needs to be solved to avoid possible MITM attacks. In terms of the evaluation, it was decided to leave these attack vectors open and observe their possible discovery by the evaluation teams.

Fusing code

Exporting the license verifications to native code opens a new requirement and measurements must be researched on how to prevent attackers from simply detaching the native code, while

[148] Notice: The NDK does not provide options for fetching these information without Java code for using the designated frameworks by Google. Therefore the information are obtained by Java code and forward to the native code via JNI.

replacing the designed getLicenseStatus() function at the same time. In the current implementation the return value is not even used. This is done to confuse attackers and the following fusing options are used instead.

Based on the general ideas of self-modifying code by other researchers (e.g., [226]), methods for Android were analyzed and implemented in [100] and as introduced in 11.4.4. Based on the sample implementation of modifying variables in memory, the idea was created to modify variables of if-statements in a running app that would usually trigger an app to quit (or other desired behaviors).

If the license reply by the Google servers is fine, the native code will edit these variables in memory to deactivate quitting so that the if-statement does not trigger any actions. Otherwise, quitting, by killing the app's process would be initiated and results in a desired behavior of null-pointer exceptions without traces to the killing of the process itself as previously outlined in 11.4.5.

In detail, the following modifications were added to the Java-based game (besides the inclusion of required native libraries / cf. nLVL guide by [62]), while providing the corresponding source code snippets in the Appendix (see 15.1.18).

- ApplicatioInfo.java
  The applicationInfo.java file comes with the nLVL source codes and supplies the methods, called by the native code, to provide user- and device information used by the nLVL [62].

- Global.java
  A condition to allow the modification of variables in memory by native code is that the desired variables have to be initialized in advance of calling the native code, while the intention was to use that method across several activities. Therefore, the idea is to use global variables. A possible way to do this on Android is the usage of Singleton Pattern as outlined in 11.4.4 and based on samples source codes by [339]. A Singleton Pattern is technically a class that, once created, can persist in memory and may be accessed from different activities. Here, the byte arrays used in later comparisons were defined, while instantiating the Single Pattern within the onCreate() method (alternatively onStart(), see 7.3.14) of the startGameActivity.java, which is the initial activity of the game checking requirements and showing the main menu to the user (more details next).

- GeoGameActivity.java
  The aforementioned global variables were used in the onResume() function in that activity to trigger a timer of 13 seconds to kill the process, if the two global variables remain identical, which occurs either upon a negative license reply or upon an attack by separating the java part from the native part.

- startGameActivity.java
  This activity takes care of checking game requirements, initializing demo limitations (with a time limit of 120 seconds), and ultimately starting up the main menu. In terms of our protection, a call via JNI to our native code was added, while providing all the Java methods called from the native code and related to the aforementioned ApplicationInfo.java that implements the provided methods in detail. For confusion, a

system property called "SystemSecure" was set to "true", but it has no function at all. Moreover, the aforementioned Singleton Pattern gets initialized, before defining two identical byte arrays (cf. "NILS2K"), and before the native code library ("myTest") is loaded and executed (see its details next). By default, an if-statement compares the byte arrays is called next, and which either kills the app process (byte arrays still equal) or does nothing (license fine & modified by native code).

- Native library myTest
  The native library myTest as shipped with the nLVL source codes [62] takes care of setting up the requirements of the nLVL library like gathering the device and user information from Java code (applicationInfo.java, see above). It also prepares the request-string (encoded URL call) for Google's license server, before initiating it by using the functions provided by the nLVL library and returning the result to the Java code. In our implementation the Java code did not use the return value and the protection relies completely on the fusing methods, which definitely confuses the attackers.

- Native library nLVL
  The nLVL [62] provides the actual functions for encoding various device- and user information into a request-string, submitting it to Google's license server (using its embedded library CURL) and taking care of the response. In terms of our fusing protection here, another function called CP() was added that performs the memory modifications upon a positive license reply. Basically it modifies any of the predefined byte arrays (namely "NILS2K", "ALLES3", "BAUM__") so that the if-statements no longer execute their malicious behaviors (e.g., kill the app process).

12.2.3 CP implementations for eval. 3 using the nLVL/heavy fusing and SignPost

In addition to the modifications presented previously in 12.2.2, the fusing methods intensified, since it was noticed in the 2nd evaluation (see 13.4) that the simulated attackers tried to attack the interfaces between native and Java code, or they looked for special strings and instructions that forced the app to quit and removed them.

Therefore, further possibilities for information exchange between native and Java code were researched and found in the environment variables. This was in addition to applying obfuscation possibilities to hide commands for quitting the app in terms of a negative license reply.

Furthermore, evaluation 3 verifies the idea of device-specific compilations using a different app-market application that gathers some device details in advance for providing the user a specially, compiled application of his desired app for that single device only. In terms of the evaluation this was emulated by fetching the details in advance and providing each student a compiled app designed for a special device only.

The source code with the following modification is found in the Appendix (see 15.1.19).

In detail the following, additional modifications were added:

- GeoGameActivity.java
  The indication to quitting the app was hidden by using the command shell for executing

the kill command of the current process instead. Furthermore the string itself (cf. "kill PID") was distributed among the whole file to hide its usage and concatenated once and a while. The used process id was gathered from a previously set global variable (cf. StartGameActivity.java) instead. Furthermore the self-defined environment variable "A_SECURE" was used to modify the path to GFX resources (used for graphics), which led to severe, non-traceable[149] issues, if it was not previously defined by the native code upon a valid license status. Another similar protection was hidden to put the used thread to sleep for several hours using the environment variable "A_WAIT", which was set upon a negative license reply. It results in an essential hang-up (blank screen) of the game and ANR[150] is not triggered, since the main-thread is not affected.

- Native library nLVL / codeinput.c
  Besides setting the environment variables "A_SECURE" and "A_WAIT" to its desired values for each use case (cf. license vs. not-licensed), the systemcall kill() was added to native code to prevent its removal from the Java source code as it was performed by attackers in the 2nd evaluation. Furthermore, two device identification methods were integrated, and compared the current device's wlan0 MAC address and serial number to the hard-coded device information. In the evaluation each app was designed for a specific device. In a real-world application this information may either be provided within the app (cf. different app market approach / see 11.2.1) or dynamically from an internet source or even a local SE.

- Native library myTest / MyTest.c
  An exported function killer() was added using the systemcall kill(). Nevertheless it is never called and is placed as a trap for confusion only.

- Global.java
  In addition to the existing information, the process-id is set upon instantiation of the Single Pattern (cf. StartGameActivity.java).

[149] to this code line
[150] ANR is some sort of crash-dialog usually triggered when the main thread is not responsive [365]

13 Evaluation / target state (security analysis)

This evaluation chapter consists of a review of selected methods from the previous proposed-solution-section that are partly used in our evaluation apps, too. Moreover, the actual results from the performed evaluations with the student testing groups are presented in this chapter.

13.1 Review of the used methods and expected protection level

While it is possible to prove the security of an encryption algorithm by estimating the time required to find the key, a difficult question arising in our research was how to prove the security increase of the presented methods. One needs to observe several factors:

- Applications are always different, and therefore methods, are hidden differently each time. Smaller applications cannot be protected as well as more complex applications and our fusing options can be hidden in more complex code much better.
- Attackers have different skills and sometimes might be an average person applying tools, while at other times, they use sophisticated reengineering equipment.
- The type of attacker also depends on the product. While less popular apps might be of interest to only smaller groups only and the attackers might have only average IT skills, the crackers that specialize in cracking software will focus only on the famous apps.
- The security of the implementation depends on the developer's skills. While this dissertation shows possible ways, it is up to the developer to integrate them safely. Even a best practice guide, as introduced earlier, is them provided.
- The chosen selection of security measurements affects the gained protection (e.g. LVL vs. nLVL).
- Ultimately, there are no figures available on the effectiveness of a method that can be used for probability calculations or risk/security analysis, while being influenced by the above factors again.

Asking an industry representative about the issue on how to prove the effectiveness and sell a security product, Thomas Goebl[151] outlined in [340] that the market regulates himself using four general guidelines, since there is no independent certification available. First of all the effectiveness is shown by how long a protection can withstand an attack ("Crack Free Window" [340]), with regard to the critical first weeks (e.g., 30 days [70]). Furthermore, the effect on the end users' experience is of importance, and in the best case, the protection has no noticeable effect. In addition, the integration factor for developers is significant and any protection needs to be applied in a reasonable amount of time, while ultimately the price for such protection is of relevance to a company applying the protection to their products. Furthermore, the VdS[152] confirmed that there are no standards or guidelines to evaluate a copyright protection [341]. Moreover, a representative by the BSI[153] confirmed that they are unable to provide guidelines as well [342].

[151] Director of Sales & Marketing, Denuvo Software Solutions GmbH, Austria
[152] The VdS is one of the leading companies known for security certification - https://www.vds.de
[153] The BSI is Germany's federal agency for security in information technology - https://www.bsi.bund.de/

Therefore, in the end it was decided to keep the currently selected way, and evaluate selected protection methods with students acting in the role of an attacker to gain details on possible attack approaches and improve the protection based on the results. The following subsections once more review the methods in general, before the evaluation is introduced and presented with its results.

13.1.1 Android is insecure

For instance, Android (verified up to 6.01) itself remains an insecure environment (see 10.1.2), and any process or data running on Android can be extracted from the internal disk (e.g., APK file with its included native libraries) or from memory (e.g., any used encryption keys used by apps or native code). Many of these actions require root rights, so we assume that almost any available device can be rooted as outlined before and any logic executed on Android itself or exchanged data is vulnerable to interception/manipulation.

13.1.2 Android app vs. native library

Nevertheless, it needs to be noted that native code that is used by native libraries, and created with the Android NDK already provides more security. Reengineering or modifying that code requires many more technical skills (see 10.3.4 and 11.4ff) than the rather simple decompilation of a usual Android app (e.g., cracking Amazon’s DRM in 10.1.4). Therefore, it was used in several solutions including the nLVL and fusing options by example.

13.1.3 Android apps vs. secure world (SE and TEEs)

While data and logic remain secure within SEs or TEEs (excluding exploit options), their interfaces to Android and any access keys or exchanged encryption keys used by the app can still be intercepted on the Android level. Due to performance reasons, outsourcing logic to SE’s provided secure world is limited, while neither do the TEEs provide a full featured Android OS (see 8.5.1, 8.5.2 and an analysis in 10.4). As described in 11.4.7 and 11.6ff, there are certain options to use a secure element for improved security, while a conceptual idea presented for TEEs (see 11.2.2) is also for additional security improvements of copyright protection.

13.1.4 Security improvements

By reviewing the previous situation of an insecure LVL and by developers not using native code at required levels to protect their apps against software piracy, this dissertation outlines several ideas (see 11 in general and extracts next) that already greatly increase the protection level. The whole situation cannot be solved completely, and Android remains an insecure operating system, while momentarily it is required to run the apps on this insecure environment at least once.

The provided nLVL solution (see 11.4.8) provides a much more secure license verification (assuming fixing the open MITM issues like integrating signature verification and proposed obfuscation methods from 11.5ff) compared to the existing Java solution of the license verification library.

In combination with methods (see 11.4.4, 11.4.5, 11.4.6) to fuse native code and Java code to make them mandatory to one another, the situation for attackers gets far more complicated and just dividing native code from the app (see 12.2.2 for implementation ideas) is no longer an option anymore. In addition, SEs (see 11.4.7 and assuming the SEAndroid issue is fixed) can be used to store license data more securely and act as some type of local license issuer, while limiting the access to servers to legitimate clients who have the required keys that the SE provided upon correct license verification only. In general, native code has been researched fairly well and there are many existing possibilities to make reengineering extremely difficult (see 11.5.3 for examples).

13.1.5 Remaining attack surfaces

The requirement to run most app parts on the insecure Android system remains an issue that cannot be fixed at the moment.

So far, there is no native Play Store alternative (see 11.2.1) nor a streamed local app solution using an Android-based TEE (see 11.2.2) available to bring desktop-level security to mobiles. It requires the contribution by the global players that hold the technical documents and resources to immensely improve that situation.

As long as the binary code of apps can be obtained, it remains vulnerable to reengineering. Known and presented obfuscation techniques can buy time to survive the critical days of a product introduction and related major sales weeks only. Ultimately, no solution is 100% secure, but it is possible to make it that difficult that we are close at that security level.

13.1.6 Protection level

As outlined at the beginning, it is not possible to categorize the protection in detail and the gained protection levels depend on the aforementioned factors. Nevertheless, based on the experimental results shown in the evaluation chapter next, there is a noticeable security gain against solutions provided by Google or Amazon. Further details are available in these chapters.

13.1.7 Comparison

While the advantages of the proposed solutions were previously introduced in prior chapters, they can be summed up by saying that they offer a much better protection against software piracy.

One of the biggest disadvantages of the currently, proposed methods related to the efforts for developers is the increased complexibility. This comes by using C/C++ language, which is one of the reasons why Google does not recommend C/C++ usage in general [329] and Java is much easier to handle and even safer in terms of memory leaks and other issues related to C/C++ development.

Nevertheless, by providing developers the libraries (nLVL) and sample source codes on request[154], it is straightforward to integrate the solution into an own project, while customizing certain aspects like placing fusing options randomly and then differently implemented in the Java code while adjusting the native source code to fit that customized protection. Moreover, general obfuscation tools still need to be applied (e.g., ProGuard). An automation of the earlier steps might be possible at the cost of customization and, is therefore, not further researched for the moment.

Another issue when developing protections using the proposed SEs, is the additional requirement of skills for that technology. While most developers are familiar with C/C++ programming and even more with Java programming, almost no one is familiar with the development of applets for SEs and its reduced functionality (cf. Javacard OS), while our access solution (cf. libaums) for earlier Android versions is also highly customized. While this fact is an advantage in terms of gained security and reengineering, it is a disadvantage for developers that are required to dive into the new topics.

Comparing these requirements to the usual LVL or even the DRM protection by Amazon for apps, we can conclude that a more secure solution requires considerably more efforts by app developers. The automatization of our approaches needs to be addressed in a separated research work and modifying code that is equally good implemented as by a real developer is not a trivial task.

Moreover, performance differences rely heavily on the actual implementation used. In our evaluation, we noticed that the startup of protected applications is a few seconds slower. This is caused by loading the native libraries, applying the changes to the running process (cf. fusing options), and immediately checking on the license. The actual amount of seconds is influenced by the Internet connection speed and in the current implementation, the nLVL will even block the execution until a reply is received.

[154] nLVL not publicly released yet due to open legal questions (cf. reengineering of Google’s services/interfaces)

13.2 Evaluation introduction

13.2.1 Attackers

The possible attackers trying to crack our apps range from sophisticated and very skilled developers (commercial crackers) to usual customers that have almost no IT knowledge such as teenagers looking on the internet for tools to circumvent protections. For instance, in 2014 [343] more than 71% of adults owned a smartphone in the US, while millennials even reach 85%. Therefore, it is fair to assume that today's customers are from all sorts of majors, having some or even no IT skills at all. Therefore, we can identify the following groups of users as possible attackers:

- non IT majors, e.g., usual customers being new to computers/smartphones
- non IT majors, with some IT skills, e.g., customers using a PC/smartphone for gaming
- non IT majors, with advanced IT skills, e.g., skilled PC/smartphone users working with them for years
- IT majors, e.g., students with profound knowledge
- IT majors, e.g., graduated students / young professionals
- IT majors, e.g., professionals
- IT majors, e.g., professionals specialized on IT security

13.2.2 Effects on the evaluation

Based on the previously, recognized attacker groups, we decided to verify the security level of different groups of attackers with different skill sets and let each group try to circumvent the developed protection, while monitoring their approaches described in detail below.

Of course, due to limited resources the results may not represent any results gathered in a large hallway test, but it certainly allows a profound assumption, if the proposed protection provides a sufficient security benefit in terms of average users / developers in general.

In addition, we verified two different kinds of protections. While our initial approach was Java-based only, our final approach of protecting an application consisted mostly of native code ideas as outlined in the proposed-solution section (see chapter 11).

The available groups in both evaluations were selected and assigned to the different attacker groups based on a questionnaire (details below). We differentiated between the following attackers:

- Beginners (Computer Science students with little Android knowledge)
- Intermediate (CS students with Android skills and few IT-security skills)
- Experts (CS students showing a good understanding of Android reengineering and IT-security in general)
- Experts 2 (same as Experts, but additionally trained with Android reengineering skills and insights to the used copyright protection solution)

13.3 Evaluation of initial (Java-based) approach

In terms of our first evaluation performed in January 2016 we had four different teams consisting of two computer science students per group as outlined before. For privacy reasons, we refer to these groups as beginners, intermediates, experts 1 and experts 2.

13.3.1 Group assignment

For determining the target group of a student, the instructor used a questionnaire form (see 15.3.1) to categorize each student in one of the three main groups (beginners, intermediates, experts). In addition, the expert 2 group was specially trained by us regarding the details of the protection, while the usual experts received a reengineering introduction only.

While the students were requested to rate their skills themselves first, the statements made were validated by viewing the answers to the questions. Based on these results it was up to the instructor's impression to categorize a student into a designated group.

13.3.2 Evaluation setup, goals and deadlines

Each team received a prepared device with the following conditions:

- Nexus 7 with Android 5.1.1
- Preinstalled, pre-activated and copyright-protected game (cf. one time license key ; see 12.2.1 for implementation details)
- Unlocked (but not rooted)

We next requested each group to try to break the protection by copying the app to another device and successfully executing the app.

Each group received a time limit of 20 hours (per student) for performing the possible attacks, while documenting each taken step in a report.

Furthermore, the expert 1 team got an introduction to Android reengineering, additionally the expert 2 team was introduced to the details of the solution (used copyright protection methods).

13.3.3 Expectations

In general, and due to the known issues with reengineering of Android Apps, we expected that the protection may be circumvented by at least one of the expert teams. The following is an overview on possible attacks/steps that could have been performed by each group in theory (marked green). Of course, we expected the more advanced groups to have the same or similar ideas like the other, lower teams (marked bright green) as shown in Table 37.

Attack / Step	**Beginner**	**Intermediate**	**Exp. 1**	**Exp. 2**
Acquire reengineering skills	🟩	🟩	🟩	🟩
Understand rough protection measurements	🟩	🟩	🟩	🟩

Discover and get APK file	■	🟩	🟩	🟩
Root device	■	🟩	🟩	🟩
Decompile APK	■	🟩	🟩	🟩
Sniff network traffic to get a deeper understanding	■	🟩	🟩	🟩
Understand copy protection in general		■	🟩	🟩
Modify decompiled code / create modified app		■	🟩	🟩
Circumvent protection (exit) on start-up		■	🟩	🟩
Modify server communication		■	🟩	🟩
Get details about exchanged information		■	🟩	🟩
Deactivate copy protection			■	■
Get decryption key by app manipulation			■	■
Sniff and attack communication			■	■
Using Xposed for attacks (network / SE)			■	■

Table 37 - Expectations on each group (based on [330, p. 23ff])

13.3.4 Results, discussion and section conclusion

Based on the previous assumptions, this section shows an overlapping with our expectations as well as additional performed steps (results) by the teams (indicated with an X), while the cracking-level is indicated, too (1 to 4, where 4 means completely cracked). If there was more than one student per level, the table shows a summary of all of them. Empty fields do not necessarily mean that the students did not perform it, but it was not mentioned in their reports.

Attack / Step	**Beginner**	**Intermediate**	**Exp. 1**	**Exp. 2**
Acquire reengineering skills	■ X	🟩 X	🟩 X[155]	🟩 X[155]
Understand rough protection measurements	■ X	🟩 X	🟩 X	🟩 X[155]
Discover and get APK file	■ X	🟩 X	🟩 X	🟩 X
Root device	■ X	🟩	🟩 X	🟩
Decompile APK	■ X	🟩 X	🟩 X	🟩 X
Sniff network traffic to get a deeper understanding	■ X	🟩	🟩	🟩
Understand copy protection in general		■	🟩 X	🟩 X[155]
Modify decompiled code / create modified app		■	🟩 X	🟩 X
Circumvent protection (exit) on start-up		■	🟩 X	🟩 X
Modify server communication		■	🟩	🟩
Get details about exchanged information		■	🟩	🟩
Deactivate copy protection			■ X	■
Get decryption key by app manipulation			■ X	■
Sniff and attack communication			■	■
Using Xposed for attacks (network / SE)			■	■
Additional actions				
obtain info / analyze SE / Papers	X			

[155] Introduced by N.T. Kannengiesser (Exp.2: by Mr. Stadler as well)

compare to open source code			X	
code adding for info. printing			X	
replacing functions / fake key			X	X
Cracking level	2	1	4	3

Table 38 – Expectations vs. Results on each group (based on [330, p. 23ff], [[286] as quoted in [330, p. 23ff]], [286])

Discussion

As expected, one of the expert teams (see [[286] as quoted in [330, p. 23ff]] and [286]) succeeded in cracking the java-based copy protection, and even used an approach that we did not initially observe, by comparing the decompiled code to the reengineered code of the open-source app, which saved them lots of time in finding the responsible functions of the copyright protection. By requesting an additional license key, they simply printed out the used encryption key to hardcode it into a modified app, besides removing any protection methods and requirements for an attached MSC. In the end, the app was also cracked and worked on another device. To our surprise the Intermediate and Exp.2 teams did not succeed in even rooting the device, which is a fundamental requirement for advanced reengineering approaches. Nevertheless, the Expert 2 team succeeded in creating a modified, still encrypted (protected) app.

Conclusion

While there might be other and even more complex java-based protections available in commercial products like DexGuard, the evaluation shows that using Java code is not the desired way of programming secure, copy-protected applications. The identified parameters may be printed out either by adding the code (or using interception frameworks such as the Xposed Framework). For another evaluation a native code approach will be used, while using a closed source application to prevent the identification of designated copy protection methods.

13.4 Evaluation of native code approaches

In terms of our second and third evaluation performed in June 2016, two different implementations were analyzed. While the first groups got an application protected by the nLVL and minor fusing options as outline in 12.2.2, the second groups were issued an even more protected application still using the nLVL, but with additional device identification routines and even more fusing options, as presented in 12.2.3.

Due to the amount of available students, the simulated attack was performed by one student per group only, while again distributing them according to their skills in different groups. For privacy reasons we will refer to these groups as beginners, intermediates, experts 1 and experts 2 on the 2nd and 3rd evaluation.

13.4.1 Group assignment

For determining the target group of a student a new question form similar to the previous one (see 15.3.2) was used to allow the instructor to categorize each student in one of the three main groups (beginners, intermediates, experts). The expert 2 group was specially trained by us about the details of the protection in addition, while the usual experts only received a reengineering introduction.

While the students were requested to rate their skills themselves first, the statements made were also validated by viewing the answers to the questions. Based on these results, it was up to the instructor's impression to categorize a student into the designated group.

For the 2nd evaluation there were five students available (2 x beginner, 1 x intermediate, 1 x expert 1, 1 x expert 2), while for the 3rd evaluation four students were distributed among the levels (1 x beginner, 1 x intermediate, 1x expert 1, 1 x expert 2).

13.4.2 Evaluation setup, goals and deadlines

2nd evaluation

In the 2nd evaluation each student (single person team) received the protected application by email, while they provided a Google account in advance. In the developer console[156] these provided accounts were added to the testing access for licensing that provided a "LICENSED"-response to each LVL requests using these accounts. Using this option it was not required (as initially thought – see question form) to buy the app. The students were offered optional rental devices, while it is not an evaluation requirement.

3rd evaluation

While the setup was similar to the aforementioned 2nd evaluation (e.g., APK file emailed), the students received a specific rental device and an app designed for that device only (cf. device identification routines, see details in 12.2.3).

Each student (single person team) received a prepared device with the following conditions:

- Nexus 5 or 7 with Android 5.1.1
- Unlocked (but not rooted)

2nd and 3rd evaluation

Next, we requested each group to try to break the protection by copying the app to another device using a different Google account and successful executing/use the app on that device.

Each group received a given time limit of 20 hours (per student) for performing the possible attacks while documenting each taken step in a report.

Furthermore, the expert 1 team got an introduction to Android reengineering. Additionally, the expert 2 team was also introduced to the details of the solution (used copy-protection).

13.4.3 Expectations

In general, and due to the known issues with reengineering of Android Apps, we expected that students would try to target the Java code, even though it required advanced security skills to

[156] See https://play.google.com/apps/publish/ → Settings → License Testing

target the native code protection. However, it offered some known flaws due to present MITM issues in the current proof-of-concept of the nLVL (see 11.4.8 for details).

The following tables are an overview on possible attacks/steps that could have been performed by each group in theory (marked green). Of course, we expected from the more advanced groups to have the same or similar ideas like the other, lower teams (marked bright green) as shown in Table 39 (evaluation 2) and Table 40 (evaluation 3).

Attack / Step	Beginner	Intermediate	Exp. 1	Exp. 2
Acquire reengineering skills	■	🟩	🟩	🟩
Acquire information on LVL by Google and others	■	🟩	🟩	🟩
Understand rough protection measurements	■	🟩	🟩	🟩
Root device	■	🟩	🟩	🟩
Decompile APK	■	🟩	🟩	🟩
Sniff encrypted network traffic	■	🟩	🟩	🟩
Understand copy protection in general		■	🟩	🟩
Modify decompiled code / create modified app		■	🟩	🟩
Circumvent a fusing method (e.g. disable exit)		■	🟩	🟩
Approached reengineering of native code		■	🟩	🟩
Understand copy protection in high detail			■	🟩
Circumvent all fusing methods			■	🟩
Using Xposed (Cydia Subs. or other) for attacks			■	🟩
Using native code overloading etc. for attacks				■
MITM / Sniff and attack LVL communication				■

Table 39 – Expectations on each group in terms of evaluation 2 (based on [330, p. 23ff])

Attack / Step	Beginner	Intermediate	Exp. 1	Exp. 2
Acquire reengineering skills	■	🟩	🟩	🟩
Acquire information on LVL by Google and others	■	🟩	🟩	🟩
Understand rough protection measurements	■	🟩	🟩	🟩
Root device	■	🟩	🟩	🟩
Decompile APK	■	🟩	🟩	🟩
Sniff encrypted network traffic	■	🟩	🟩	🟩
Understand copy protection in general		■	🟩	🟩
Modify decompiled code / create modified app		■	🟩	🟩
Circumvent a fusing method (e.g. disable exit)		■	🟩	🟩
Approached reengineering of native code		■	🟩	🟩
Understand copy protection in high detail			■	🟩
Using Xposed (Cydia Subs. or other) for attacks			■	🟩
Circumvent all fusing methods				■
Circumvent device identification methods				■

Using native code overloading etc. for attacks				🟩
MITM / Sniff and attack LVL communication				🟩

Table 40 – Expectations on each group in terms of evaluation 3 (based on [330, p. 23ff])

13.4.4 Results, discussion and section conclusion

Based on the previous assumptions, this section shows an overlapping with our expectations as well as additional performed steps (results) by the teams (indicated with an X), while the cracking-level is indicated, too (1 to 4, where 4 means completely cracked). If there was more than one student per level, the table shows a summary of all of them. Empty fields do not necessarily mean that the students did not perform it, but it was not mentioned in their reports.

Attack / Step	**Beginner**	**Intermediate**	**Exp. 1**	**Exp. 2**
Acquire reengineering skills	X	X	X[157]	X[157]
Acquire information on LVL by Google and others	X	X	X[157]	X[157]
Understand rough protection measurements	X	X	X	X[157]
Root device	🟩	🟩	🟩	🟩
Decompile APK	X	X	X	X
Sniff encrypted network traffic	🟩	🟩	🟩	🟩
Understand copy protection in general		🟩	X	X
Modify decompiled code / create modified app	X	X	X	X
Circumvent a fusing method (e.g. disable exit)	X	🟩	X	X
Approached reengineering of native code	X	X	X	X
Understand copy protection in high detail			🟩	🟩
Circumvent all fusing methods	X[158]	X[159]	X	X
Using Xposed (Cydia Subs. Or other) for attacks			🟩	X
Using native code overloading etc. for attacks				🟩
MITM / Sniff and attack LVL communication				🟩
Additional actions				
Acquire Java source codes (dex2jar or other tool)	X		X	X
Tried to analyze obfuscated Java codes	X		X	X
Tried decoupling Java/Native code	X			
Looked for values & string (e.g. 120 / exit calls)	X		X	
Tried cracking tools (e.g. AntiLVL)		X		
Used insight knowledge to find fusing methods				X

[157] Introduced by N.T. Kannengiesser
[158] After hint that the app still terminates after 13 seconds (not counting – severe cracking level)
[159] Removed timer tasks to remove demo limitation and 2nd fusing option (timed exit) apparently by accident

Cracking level	3	4	4	4

Table 41 – Expectations vs. Results on each group in terms of evaluation 2 (based on [330, p. 23ff] and [284])

Attack / Step	**Beginner**	**Intermediate**	**Exp. 1**	**Exp. 2**
Acquire reengineering skills	X	X	X[160]	X[160]
Acquire information on LVL by Google and others		X	X[160]	X[160]
Understand rough protection measurements	X	X	X	X[160]
Root device				
Decompile APK	X	X	X	X
Sniff encrypted network traffic				
Understand copy protection in general				X
Modify decompiled code / create modified app	X	X	X	X
Circumvent a fusing method (e.g. disable exit)	X	X	X	X
Approached reengineering of native code	X			X
Understand copy protection in high detail				
Using Xposed (Cydia Subs. Or other) for attacks				
Circumvent all fusing methods				
Circumvent device identification methods				
Using native code overloading etc. for attacks				
MITM / Sniff and attack LVL communication				
Additional actions				
Acquire Java source codes (dex2jar or other tool)	X	X		X
Tried to analyze obfuscated Java codes	X	X		
Tried decoupling Java/Native code	X	X		X
Looked for values & string (e.g. 120 / exit calls)	X			X
Tried cracking tools (e.g. AntiLVL)	X	X		
Used insight knowledge (sniff network traffic)				X
Tried DeObfuscation tool	X			
Using advanced tools gdb, IDA, other disassem.				X
Replacing OP codes in native code				X
Tried to sniff traffic using proxy app (no success)				X
Cracking level	1	1	1	1

Table 42 – Expectations vs. Results on each group in terms of evaluation 3 (based on [330, p. 23ff], [285])

[160] Introduced by N.T. Kannengiesser

Discussion

- 2nd Evaluation

  As expected and written in [284], the students mainly tried to target the Java code (and the fusing options) and identification was fairly easy due to the used exit calls. Some may not have completely understood its logic and relation to native code, describing the if-statement of comparing two identical byte arrays as “unnecessary or suspicious” [284], but they removed the exit calls. That disabled the protection, of course. Nevertheless, they had severe issues with the native code commenting that it was “time-consuming and nerve-wrecking […] [and] only clues [are] […] imported Java or Android libraries” [284]. Even the expert teams (here expert 2), who made the most progress on analyzing it by identifying the performed memory modification by the native code, were not able to deactivate the nLVL itself within the given time frame of 20 hours. The results were observed for our third evaluation making the fusing options harder to discover than ever, besides the usage of further protections methods (cf. device identification).

- 3rd evaluation

  As assumed and written in the reports [285] the students had significant problems in discovering all used fusing options, while still focusing mainly on the Java code. Nevertheless, they were able to discover one of these fusing options in most cases, but they were not able to prevent the application-killing executed by the native code or Java code upon negative license reply, or issued by the Java code when a decoupling (= attempted cracking) took place. Since the nLVL uses its own network libraries (CURL), it remains immune to the performed proxy ideas as well and only sophisticated MITM attacks could have been a threat due to the existing and known limitations in the current proof-of-concept of the nLVL (cf. missing signature verifications or HTTPS certificate verification). Moreover, the inclusion of faked functions like the killer() one in myTest.c proved to be a good (time-consuming) idea. In general, using native code had the desired effect and students described it by saying “we weren’t able to identify how the code work [sic!] because the assembly was very hard to understand” [285] and “Immerhin scheint der Schutz auf jeden Fall wirkungsvoller zu sein, als der Vergleichsschutz aus der Amazon DRM, was ein aussagekräftiges Statement ist”[161] [285], even our expert 2 team used various disassemblers including top-notch tools such as IDA and other sophisticated disassemblers. All of them failed in pirating the app.

Conclusion

While our initial ideas and Java-based approaches were still vulnerable to attacks, the more our solutions moved to the native code versions, we were able to notice an increase in the protection of our solution.

[161] transcript by author: At least, the used protection seems more effective than the protection provided by Amazon DRM, which is a meaningful statement

For instance, in the 2nd evaluation the protection was still largely based on Java code (cf. few fusing options), while the native code handled the licensing only.

Instead, in the 3rd evaluation, the fusing between native code and Java code got intensified, increasing the protection quite a lot and ultimately protecting the app within the desired time frame. While the students were able to discover a fusing option, they were not able to identify the deeply hidden ones using indirect method triggering by using the environment variables for the transport of messages. Of course, that is a current idea and attackers may observe this option in the future. Nevertheless, it illustrates the need to increase the research on this topic, while the current research work can only act as an introduction to this.

Moreover, the current small evaluation group of computer science students that specialized on Android may not reflect a huge hall way test due to our limited resources. Nevertheless, it still underlines our assumptions that using native code provides a real benefit for the security of Android applications executed in an insecure environment.

“You can make hamburgers with a cow, but you can't make a cow with hamburgers.” [344]
Carlos Gutierrez

(on the issue of reengineering of native code)

14 Summary

14.1 Review research questions

Before summarizing the results in the conclusion below, the earlier research questions should be reviewed, once again as well:

No.	Question
0	Fundamental question: Are the current copyright protections for Android sufficiently secure?
1	If that is not the case, how can we ensure that an app is used on a valid device or by the valid user only?
2	Is it possible to store sensitive information like licensed data more securely, maybe, e.g., by using a secure element or alternatives?
3	Is it actually possible to use a secure element on Android (as a developer)?
4	How can we improve copyright protections and how can we implement them on Android?
5	How can we protect apps against reengineering (cf. static- and dynamic analysis) and is that actually possible with usual Android versions?
6	Might it be a better approach to use native code for security related issues instead of Java (cf. desktop world is dominated by native code and iOS uses it as well)?
7	What needs to happen elsewhere to improve the situation, (e.g., hardware modification and/or better cooperation by different manufacturers)?

No.	Short Answer	Details
0	No	Here it needs be outlined that other researchers investigated the LVL in 2010 already [320], while investigations on the Google's LVL and Amazon's DRM (besides other Java solutions) performed in this research confirmed their earlier findings that the protection is severely broken. Even additional solutions (cf. third party research) exists, they are not used by major app markets.
1	Many options	There are various options like integrating user/device attributes, storing information on SE or using native code safely, besides using several native code protections (see chapter 11 for details and related question answers below).
2	Yes	Besides using native code and (native) file encryption that can be protected much better from reengineering (see analysis in 10.3), using a SE is an option when certain issues are solved (11.4.7).
3	Partly	As outlined in 11.4.7 there are remaining, secure options, but due to enforced SEAndroid in recent versions, a secure, native code version cannot be used right now. That limits the possibilities and a Java version is vulnerable to the severe reengineering issues (cf. 10.1ff), with limited solutions left (cf. 11.4.7).
4	→	Using native code is more secure than using Java code (see chapter 11 for details and related question answers below).

5	Partly	It is not possible to prevent reengineering in total (exceptions: 11.5.1), since Android remains an insecure OS (see 10.1.2). Nevertheless comparing the reengineering of Java code with the problems that attackers have when reengineering native code (cf. 13.4), we found sufficiently secure solutions to that issue.
6	Yes	Definitely. While we already assumed an increased protection in our analysis of native codes (see 10.3), the simulated attackers were not able to circumvent the native solutions (see 13.4.4) and outlined their severe issues with it.
7	→	The best solution would be to get rid of DEX code that is very vulnerable to reengineering. Unfortunately that is a major task and requires support by Google due to fundamental VM changes, while there is almost no documentation available. Furthermore hardware manufactures can cooperate with Google to provide users/developers access to SE/TEEs (see 11.2 for details).

14.2 Contributions

This section presents our own contributions and outcomes of this research work, while it does not cover any third party solutions that may have been mentioned in the solution sections. Our contributions are separated into different stages to allow a greater overview.

Confirming the assumed issues (the problem statement)

- We were able to confirm the severe reengineering issues (cf. LVL cracking by others [320]) that apply to DEX files and any Android version in general, while showing that it applies to other protection solutions, e.g., by Amazon (see 10.1.4) and outlining even more advanced, universal cracking solutions (see 10.1.5). Moreover, we revealed the procedures of commonly used cracking solutions in high detail (see 10.1.3) to allow developers an insight and possible solutions (see 11.4).
- Furthermore we analyzed Android itself and can confirm the intense insecurity of the system due to rooting possibilities for any version (see 10.1.2), which highly affects copyright protection solutions and puts them at danger.

Confirming the security gain

- By using a sophisticated decompiler of another dissertation, we showed the security gain by using native code and comparison of the reengineering results (see 10.3ff). Moreover, assumptions that typical, future developers (computer science students) are not familiar with native code and especially ARM assembly, were confirmed by conducted surveys (see 10.3.5).

Implementing possible solutions

- Based on the results of a more secured native code, a native version of the LVL called nLVL was developed (see 11.4.8) and new issues (cf. “How to prevent the simple separation of the native part?”) addressed by introducing options like “code fusing” and “indirect method triggering” (see 11.4.6 and 11.4.5) to counteract that issue, while researching required foundations for live process modification on Android in advance (see 11.4.4).

- Furthermore, options to load code dynamically in Java and native code were shown. A special solution is loading native code from memory (see 11.4.3) that allows developers to hide simple actions even better.

Showing further possible solutions (conceptual)

- Since the secure implementation of using SEs by using native code was not possible at one point, several ideas were shown as conceptual only (see 11.4.7 and 11.6).
- Due to unknown details of the ART VM as well as implementation requirements that are out of scope for this work, an idea for the realization of a native and even more secure app store was introduced (see 11.2.1) requiring the participation of Google for its actual realization.
- Partly implemented by assuming the existence of the aforementioned native app store, options for the device- and user identification to be integrated as part of a copyright protection were analyzed and outlined (see 11.4.1).
- Moreover, the same approach can be used to prevent piracy by allowing the identification of APK's owners by the embedded user attributes (see 11.5.4).
- In addition, an even more secure idea of streaming an apps' UI for an ultimate copyright protection using a TEE was shown (see 11.2.2) and requires modifications to Android and a TEE to execute apps. Here, a cooperation of Google and a device manufacturer is required.

Verification of the expected security gain

- The proposed, implemented solutions were reviewed (see 13.1), adapted to sample implementations, and applied to a usual Android apps (see 12), for demonstration and verification purposes. The desired security increase was confirmed and simulated attackers did not succeed in breaking the final, native protection (see 13.4).

14.3 Conclusion

At the beginning, the topic was defined by investigations into the security of Android that proved to be quite insecure thinking about app-, data-, and license protection (see 10.1.2, 10.1.3, 10.1.4 and 10.1.5).

Therefore, the goals of this dissertation were defined to identify ways to improve available copyright protection mechanisms that increase the difficulties on reengineering and introduce (mainly) developers to the necessary skills and methods to avoid the most prevalent and common issues by outlining the problems and several examples to picture the current situation in a detailed analysis (see chapter 10) and to provide solution ideas (see chapter 11). The fact that the nLVL idea was realized by reengineering Google's frameworks underlines the severe security issues on Android once more.

In addition, ideas for the global players were identified that cannot be realized by usual app developers (this author) without the help and services of companies like Google that provide the manpower and missing knowledge on undocumented features (see 11.2ff).

In general, it is not possible to avoid reengineering in total, and therefore, an attack surface always remains, even if the shown approaches try to lower the risks. In summary (see 11.4ff for all details), these are methods to include user and device attributes for recognizing valid users and devices, besides ideas to store data in a more secure manner, while showing possibilities for modern Android versions to load additional program parts dynamically again. Moreover, methods to manipulate the app process' memory and use it in terms of the copyright protection to bind program parts (here Java and native code) more securely together were presented, and suggested solutions like a proof-of-concept of a native version from Google's LVL for gaining additional security was shown. In addition, ideas to use SE for copyright protection were presented as well (11.6ff), even though most of the ideas cannot be realized due to access restrictions by SEAndroid (cf. 11.4.7).

Unfortunately, one must assume that the situation for using secure native code gets even worse, since Google is about to enable even further restrictions on the NDK with its upcoming N release [311] and so far – to the author's knowledge - without presenting adequate copyright protection solutions. For the sake of completeness, further methods by third parties were presented (see 11.5ff) as well as general solutions for protecting Java- and native code.

As pointed out in the 3rd evaluation (see 13.4.4) that included most of our proposed methods to protect an application from being copied, all evaluation groups were not able to circumvent the most recent protection. This included computer science students with some Android experience as well as those that were rated at an expert level and even specially trained by us on the used methods. They all tried to attack the protection with no success. Therefore, one can fairly assume that the proposed methods are sufficiently secure to protect apps against usual customers trying to circumvent the protection, while one can also assume that even more skilled customers with certain IT skills will have issues on cracking the protection. Furthermore, the presented solutions are meant for Android only, as even the title suggests more general solutions.

In summary, we were able to discover methods to improve the current situation, but there is still space for further improvements that requires the help and cooperation of global players like Google and the device manufacturers (see 11.2.1 and 11.2.2).

14.4 Future work

Obviously implementing copy protection for Android remains an unfinished task and while this dissertation outlines several solution approaches, we discovered new issues that may be addressed in the future.

For instance, an interesting way for protecting code is dynamic code loading after the license is verified. Unfortunately, Google limited the options with the introduction of the ART VM as outlined earlier (see 11.4.3 for details). Even we were able to find adequate solutions for native code instead. However, the execution of dynamically loaded native code is currently highly limited to simple tasks due to missing linker functionalities in our current implementation.

Therefore, investigating solutions for providing that capability can be researched even further, by combining them with the ideas of [223] to allow the secure loading of external code in addition.

Moreover, it needs to be noted that many of the current issues with copyright protection on Android may be fixed by using Trusted Execution Environments (TEE) in the future. They limit the access rights of apps in addition to granting exclusive hardware access for privileged apps [217] to exchange data with the customer in an isolated manner [202]. Early, conceptual ideas were presented in 11.2.2. The downside is that it requires new hardware or system software at least, and development is limited by the costs, while Android development in general and our presented solutions are freely available. At a recent conference [345], the Trustonic Company announced that, e.g., their TEE solution is already available on a major number of Samsung devices, but it is not available on all devices worldwide.

In one of its recent releases (e.g., Android KitKat) Google tried to implement some security features that act similar to TEE (e.g., limit file access), but it is implemented in software only using SELinux [346] and cannot completely solve the issues discussed above due to possible exploits in hardware or software. Originally SELinux (aka SEAndroid) was developed by the NSA [103]. Unfortunately, SEAndroid's enforced policies are the major reasons that the usage of external SEs is not securely possible (cf. native code) these days either. While there seems to be no immanent solutions to that issue, the libUSB team may address that issue in the future, allowing the presented, conceptual ideas (see 11.6) to be implemented in a more secure manner, since the realization as a Java version is senseless and it does not provide hardly any security benefit.

Furthermore, the research by third parties to protect and hide data in the processor cache by Patrick Colb et al. [227] could be researched in terms of copyright protection to hide and encrypt data even further.

Another technology coming up soon is Project ARA by Google with shipments of early developer editions in fall 2016 [333]. While demo applications of higher-priced apps may still be available on app stores, the idea could be to use (to be developed) memory modules with integrated and performant SEs to gain additional security, outsourcing program logic to it to make it a mandatory requirement to have a physical module. It is similar to the cartridges known from manufactures like Nintendo for their devices. Furthermore, more generalized protections may be provided by allowing others to use a security module to outsource code parts.

In addition, the identified methods for copyright protection (cf. user and device identification) can be used to address future license options (see 11.3 for a general overview of options), since Android is just emerging to desktop computers (cf. Remix OS[162]) and typical licenses addressing many devices and one user only (current mostly used license option) have to be adapted to allow volume licenses. Typical volume licenses allow the installation on several devices and are used by various users. These cases cannot be addressed with the currently available methods and Google's and Amazon's solutions are meant to be used with one account and one user (maybe his family) only.

Due to our discoveries in terms of attacking libraries (cf. LVL), it is recommended to review other related frameworks. For instance, the in-app-billing libraries by Google should be

[162] http://www.jide.com/remixos

carefully reviewed, since they work quite similar to the LVL. In fact, in-app-billing has been circumvented in a research performed by [248] already. It is certainly preferred to port it to a native version as well.

14.5 Legal

All presented information may be used for research purposes only.

The goal of this work is to raise the awareness of industry and developers for the security issues related to copyright protection on Android.

Months in advance of the release of this dissertation and following a guideline by Google that 90 days are sufficient to fix any issues [347], we informed all affected companies, which are in charge of the used protections by thousands of developers, of our findings.

These global players need to act and raise the awareness of the issues at least, and preferably provide developers even better solutions in the future. Our results could be the base for these solutions.

Notified companies

We notified **Google** about the issues with the LVL by September 2014 and they classified it in their reply "as a low security issue since it requires the device to be rooted" [348]. Furthermore, we also provided them information about our native LVL approach by May 2016 and invited them to participate.

We also notified **Amazon** about possible issues with their DRM protection by February 2016 [349]. In further discussions with their security team, it turned out that the protection was already modified, but we were able to circumvent it once more and provided Amazon additional information in June 2016.

Samsung was notified about issues with their Zirkonia library by April 2016 [350].

Further companies that received notifications about issues by us as well as early suggestions were BMW (2013) and Team17 (June 2016).

NDA exceptions

The presented information about the used and presented MSC by Giesecke & Devrient in section 8.5.1 was verified and approved by Dr. Sterzinger for publication.

15 Appendix

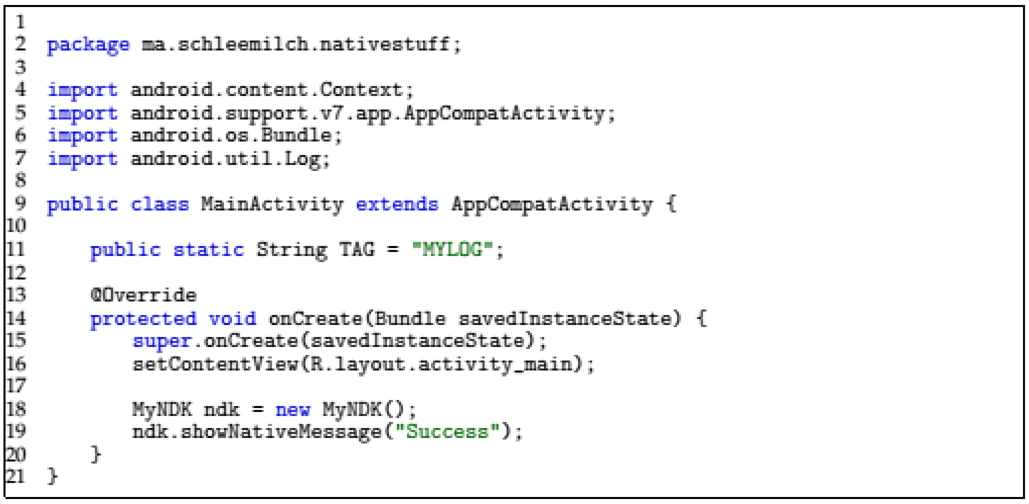
15.1 Source codes

15.1.1 Code for intercepting LVL and manipulating license response

Due to the fact that these source codes may be misused, they are available by request only and internally accessible in the unpublished thesis of Mr. Marius Muntean [64, pp. 58,59].

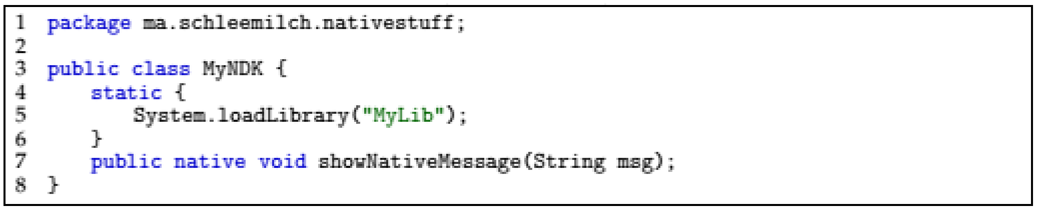
15.1.2 Android project with native code (Android Studio)

The following source examples illustrate the basic integration of native code into an Android Studio project. Further information may be found in the fundamental section on p. 43.



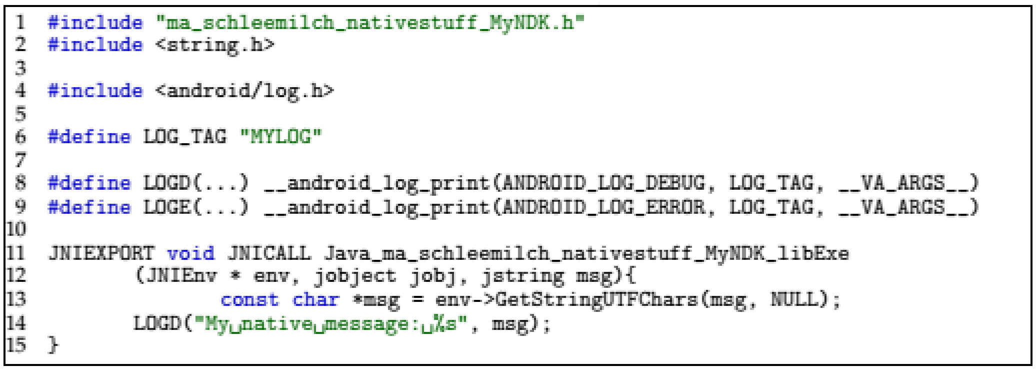
```java

package ma.schleemilch.nativestuff;

import android.content.Context;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity {

    public static String TAG = "MYLOG";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        MyNDK ndk = new MyNDK();
        ndk.showNativeMessage("Success");
    }
}
```

Figure 75 – MainActivity.java /Main Activity of the example project [100, p. Appendix A]



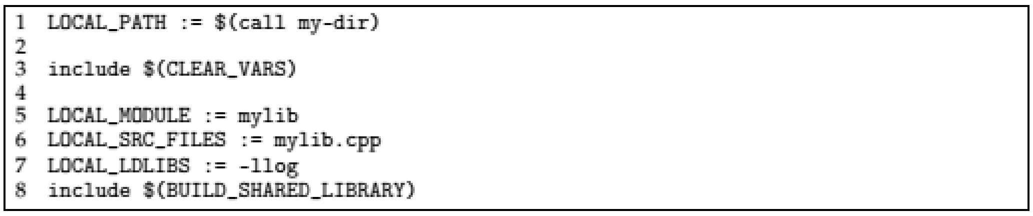
```java
package ma.schleemilch.nativestuff;

public class MyNDK {
    static {
        System.loadLibrary("MyLib");
    }
    public native void showNativeMessage(String msg);
}
```

Figure 76 - MyNDK.java / Class for providing native code [100, p. Appendix A]



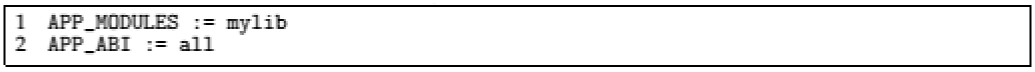
```
#include "ma_schleemilch_nativestuff_MyNDK.h"
#include <string.h>

#include <android/log.h>

#define LOG_TAG "MYLOG"

#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

JNIEXPORT void JNICALL Java_ma_schleemilch_nativestuff_MyNDK_libExe
        (JNIEnv * env, jobject jobj, jstring msg){
                const char *msg = env->GetStringUTFChars(msg, NULL);
        LOGD("My␣native␣message:␣%s", msg);
}
```

Figure 77 - mylib.cpp / C++ file with native code implementation [100, p. Appendix A]



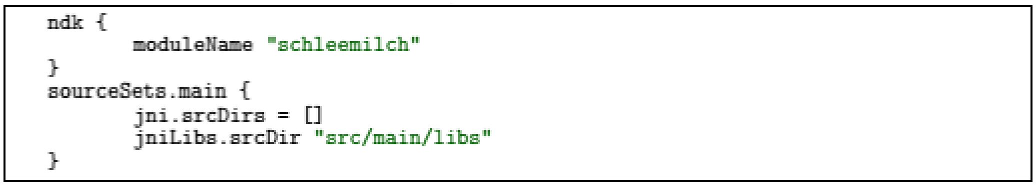
```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := mylib
LOCAL_SRC_FILES := mylib.cpp
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

Figure 78 - Android.mk / Android Makefile [100, p. Appendix A]



```
APP_MODULES := mylib
APP_ABI := all
```

Figure 79 Application.mk / Application Makefile [100, p. Appendix A]



```
ndk {
        moduleName "schleemilch"
}
sourceSets.main {
        jni.srcDirs = []
        jniLibs.srcDir "src/main/libs"
}
```

Figure 80 - build.gradle / Gradle Build File [100, p. Appendix A]

15.1.3 Simple JNI Code Sample and its decompiled source code

```
#include <com_example_testnative_TestLib.h>

JNIEXPORT jlong JNICALL Java_com_example_testnative_TestLib_testNativeValue
  (JNIEnv *env, jclass clazz, jlong b) {
        jlong a = 20 * b;
        return a;
}
```

Table 43 - Simple C code to be used within the Android NDK [106]

```
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2016 Retargetable Decompiler <info@retdec.com>
// modified by N.T. Kannengiesser and reduced to important functions (36
// of 2175 lines); original binary file was compiled with -O3 and no
// symbols by Android NDK
```

```
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

// ---------------- Integer Types Definitions -----------------
// [...]

// ----------------- Float Types Definitions ------------------
// [...]

// ------------------------ Structures ------------------------
// [...]

// ------------------- Function Prototypes --------------------
// [...]

// --------------------- Global Variables ---------------------
// [...]

// ------------------------ Functions -------------------------

// Address range: 0xc18 - 0xc27
int32_t Java_com_example_testnative_TestLib_testNativeValue(int32_t a1) {
    // 0xc18
    return 20 * g15;
}

// [...]
```

Table 44 - Modified and decompiled example code of Table 43 using -O3 and no symbols upon compilation by Android NDK (based on output of [351])

15.1.4 Small C program and its decompiled source code

```
#include <stdio.h>

long func(long value);

int main() {
    printf("%ld\n", func(20.00));
    return 0;
}

long func(long value) {
        long a = 20 * value;
        return a;
}
```

Table 45 - Simple C code example

```
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2016 Retargetable Decompiler <info@retdec.com>
// modified by N.T. Kannengiesser and reduced to important functions (45
// of 189 lines); original binary file was compiled with -O3 by retdec

//
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2016 Retargetable Decompiler <info@retdec.com>
//

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

// ------------------- Function Prototypes --------------------

int32_t __do_global_dtors_aux(int32_t a1);
void __libc_csu_fini(void);
int32_t __libc_csu_init(int32_t result, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t
a6, int32_t a7, int32_t a8, int32_t a9, int32_t a10);
void _fini(void);
void _init(void);
int32_t _start(int32_t a1);
void call_weak_fn(void);
void deregister_tm_clones(int32_t a1);
int32_t frame_dummy(int32_t a1, int32_t a2, int32_t a3, int32_t a4);
int32_t func(void);
int32_t function_845c(int32_t a1);
int32_t register_tm_clones(int32_t a1);
int32_t unknown_83d4(void);

// ------------------------ Functions -------------------------

// Address range: 0x83ec - 0x8407
int main(int argc, char ** argv) {
    int32_t v1; // 0x10650
    char v2; // 0x10774
    // 0x83ec
    printf("%ld\n", 400);
    return 0;
}

// Address range: 0x853c - 0x8547
int32_t func(void) {
    int32_t v1; // 0x10650
    char v2; // 0x10774
    // 0x853c
    int32_t v3;
    return 20 * v3;
}
[...]
```

Table 46 - Modified and decompiled code of Table 45 using -O3 upon compilation (based on output of [351])

15.1.5 Conversion of code using control flow flattening and instruction substitution

Control flow flattening	
Original Code	Modified Code using Control Flow Flattening

```
#include <stdlib.h>
int main(int argc, char** argv) {
  int a = atoi(argv[1]);
  if(a == 0)
    return 1;
  else
    return 10;
  return 0;
}
```
	```
	#include <stdlib.h>
	int main(int argc, char** argv) {
	  int a = atoi(argv[1]);
	  int b = 0;
	  while(1) {
	    switch(b) {
	      case 0:
	        if(a == 0)
	          b = 1;
	        else
	          b = 2;
	        break;
	      case 1:
	        return 1;
	      case 2:
	        return 10;
	      default:
	        break;
	    }
	  }
	  return 0;
	}
	```

Table 47 - Example for Control flow flattening performed by Obfuscator-LLVM (based on [352])

Instructions Substitution	
Original Code	Modified Code using Instructions Substitution

Addition
a = b + c;
	a = b - (-c);
	
	OR
	
	r = rand (); a = b + r; a = a + c; a = a – r;

Table 48 - Example for Instruction Substitution performed by Obfuscator-LLVM (based on [353])

15.1.6 Example source code and decompiled code protected by Obf.-LLVM

```
#include <stdio.h>

int main(void){
printf("Hello, world\n");
return 0;
}
```

Table 49 - Simple C source code example to be used with Android NDK and Obfuscator-LLVM [279]

```
//
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2016 Retargetable Decompiler <info@retdec.com>
// Comment by N.T. Kannengiesser: used binary taken from [279]

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

// ------------------- Function Prototypes --------------------

int32_t entry_point(void);
void function_8284(void);
void (*function_8338(void))();
int32_t function_834c(char * str, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6,
int32_t a7);

// ------------------------ Functions -------------------------

// Address range: 0x8284 - 0x82bb
void function_8284(void) {
    int32_t v1 = 0; // 0xa000
}

// Address range: 0x82bc - 0x8337
int32_t entry_point(void) {
    int32_t v1; // 0xa000
    __libc_init();
    int32_t v2;
    return &v2;
}

// Address range: 0x8338 - 0x834b
void (*function_8338(void))() {
    // 0x8338
    int32_t v1; // 0xa000
    return (void (*)())__cxa_atexit(NULL, NULL, (char *)&v1);
}
```

```
// Address range: 0x834c - 0x8363
int32_t function_834c(char * str, int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6,
int32_t a7) {
    int32_t v1; // 0xa000
    // 0x834c
    return puts(str);
}

// --------------- Dynamically Linked Functions ---------------

// int __cxa_atexit(void(*func)(void *), void * arg, void * dso_handle);
// void __libc_init(void);
// int puts(const char *);

// --------------------- Meta-Information ---------------------

// Detected compiler/packer: gcc (4.6)
// Detected functions: 4
// Decompiler release: v2.1.2 (2016-01-27)
// Decompilation date: 2016-03-08 14:02:05
```

Table 50 - Decompiled version of the source code of Table 49 (output of [351])

15.1.7 Example source code using JNI and its corresponding decompiled code

```
/**
 * Created by nils on 22.03.2016.
 * [...]
 */
public class Account {

    public Account(){
    };
    public String getUsername(Context c) {
        // [...]
        return "Nils-Teststring";
    }
}
```

Table 51 - Account class to be called from native code

```
/**
 * Created by nils on 22.03.2016.
 * code based on source below
 * [...]
 */

JNIEXPORT jstring JNICALL
Java_com_example_nils_myapplication_MyNDK_getMyString
```

```
        (JNIEnv * env, jobject thiz, jobject thiz2) {

 const char *str;

 jclass myclass_class =(jclass) env->NewGlobalRef
        (env->FindClass ("com/example/nils/myapplication/Account"));

 jmethodID constructorID = env->GetMethodID
        (myclass_class, "<init>", "()V");

 jmethodID methodID = env->GetMethodID
        (myclass_class, "getUsername", "(Landroid/content/Context;)Ljava/lang/String;");

 jobject myclass_object =  env->NewObject
        (myclass_class, constructorID);

 jstring s = (jstring)  env->CallObjectMethod
        (myclass_object, methodID, thiz2);

 str = env->GetStringUTFChars(s, 0);
 __android_log_print(ANDROID_LOG_ERROR,"NATIVE-CODE", "str %s", str );

 env->ReleaseStringUTFChars(s, str);
 return s;
}
```

Table 52 - Native code that calls the Java function getUsername (based on [354])

```
// This file was generated by the Retargetable Decompiler
// Website: https://retdec.com
// Copyright (c) 2016 Retargetable Decompiler <info@retdec.com>
// modified by N.T. Kannengiesser and reduced to important functions (120
// of 3022 lines); original binary file was compiled with -O3 by Android NDK

#include [...]

// ----------------- Float Types Definitions ------------------
[...]

// ------------------------ Structures ------------------------
[...]

// ------------------- Function Prototypes --------------------

int32_t _ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz(struct struct_1 a1,
int32_t a2, int32_t a3, int32_t a4, int32_t a5, uint32_t a6, int32_t a7, int32_t a8, int32_t a9,
int32_t a10, int32_t a11);
int32_t _ZN7_JNIEnv9NewObjectEP7_jclassP10_jmethodIDz(struct struct_1 a1, int32_t
a2, int32_t a3, int32_t a4, int32_t a5, uint32_t a6, int32_t a7, int32_t a8, int32_t a9, int32_t
a10, int32_t a11);
// [...]  around 80 function prototypes
```

```
int32_t function_f48(void);
int32_t function_fec(int32_t a1);
int32_t Java_com_example_nils_myapplication_MyNDK_getMyString(struct struct_1 a1,
int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6, int32_t a7, int32_t a8, int32_t a9,
int32_t a10, int32_t a11, int32_t a12, int32_t a13, int32_t a14, int32_t a15);
int32_t unknown_1928(void);
// [...] 6 function prototypes with unknown*

// ------------------------ Functions -------------------------
// Address range: 0xd78 - 0xdab
// Demangled:     _JNIEnv::NewObject(_jclass *, _jmethodID *, ellipsis)
int32_t _ZN7_JNIEnv9NewObjectEP7_jclassP10_jmethodIDz(struct struct_1 a1, int32_t
a2, int32_t a3, int32_t a4, int32_t a5, uint32_t a6, int32_t a7, int32_t a8, int32_t a9, int32_t
a10, int32_t a11) {
// [...] looks similar to the function below
    return result;
}

// Address range: 0xdac - 0xddf
// Demangled:     _JNIEnv::CallObjectMethod(_jobject *, _jmethodID *, ellipsis)
int32_t _ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz(struct struct_1 a1,
int32_t a2, int32_t a3, int32_t a4, int32_t a5, uint32_t a6, int32_t a7, int32_t a8, int32_t a9,
int32_t a10, int32_t a11) {
    int32_t v1;
    struct struct_1 v2; // bp+10
    struct struct_1 v3; // bp+14
    int32_t (*v4)[2]; // bp+38
    int32_t v5; // 0x4000
    int32_t v6;
    struct struct_1 v7;
    struct struct_1 v8;
    int32_t v9;
    int32_t v10;
    int32_t v11;
    // 0xdac
    int32_t v12;
    ((int32_t (*)())(v12 & -2))();
    int32_t v13;
    if (v13 != a6) {
        // 0xdd0
        function_1d10();
        // branch -> 0xdd4
    }
    // 0xdd4
    ((int32_t (*)())(a10 & -2))();
    int32_t result;
    return result;
}

// Address range: 0xde0 - 0xe93
```

```
int32_t Java_com_example_nils_myapplication_MyNDK_getMyString(struct struct_1 a1,
int32_t a2, int32_t a3, int32_t a4, int32_t a5, int32_t a6, int32_t a7, int32_t a8, int32_t a9,
int32_t a10, int32_t a11, int32_t a12, int32_t a13, int32_t a14, int32_t a15) {
    struct struct_1 v1; // bp+14
    struct struct_1 v2; // bp+14
    int32_t v3;
    struct struct_1 v4; // bp+10
    struct struct_1 v5; // bp+14
    int32_t (*v6)[2]; // bp+38
    int32_t v7; // 0x4000
    struct struct_1 v8;
    struct struct_1 v9;
    int32_t v10;
    int32_t v11;
    int32_t v12;
    // 0xde0
    int32_t v13;
    int32_t v14 = v13;
    int32_t v15;
    int32_t v16 = v15;
    v13 = a1.e0;
    int32_t v17;
    ((int32_t (*)())(v17 & -2))();
    int32_t v18;
    ((int32_t (*)())(v18 & -2))();
    int32_t v19;
    ((int32_t (*)())(v19 & -2))();
    int32_t v20;
    ((int32_t (*)())(v20 & -2))();
    v15 = v13;
    v1 = (struct struct_1){
        .e0 = 0,
        .e1 = 0
    };
    v1.e0 = v13;
    int32_t v21;
    int32_t v22;
    int32_t v23;
    int32_t v24 = _ZN7_JNIEnv9NewObjectEP7_jclassP10_jmethodIDz(v1, v13, v13,
0x2042, v21, a3, a3, v14, v22, v16, v23);
    v2 = (struct struct_1){
        .e0 = 0,
        .e1 = 0
    };
    v2.e0 = v13;
    int32_t v25;
    _ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz(v2, v24, v15, a6, v25,
a3, a3, v14, v22, v16, v23);
    _ZN7_JNIEnv16CallObjectMethodEP8_jobjectP10_jmethodIDz(v2, v24, v15, a6, v25,
a3, a3, v14, v22, v16, v23);
    int32_t v26;
    ((int32_t (*)())(v26 & -2))();
    int32_t v27;
    function_1d20(v27);
    int32_t v28;
    ((int32_t (*)())(v28 & -2))();
    ((int32_t (*)())a12)();
    int32_t v29;
    return v29 / 256;
}
```

Table 53 - Decompiled version of the source code of Table Table 52 (output of [351])

15.1.8 Example for intercepting a library method using LD_PRELOAD directive

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE    := nativeHook

LOCAL_CFLAGS += -O3 -fPIC

LOCAL_SRC_FILES := nativeHook.c

LOCAL_C_INCLUDES := $(LOCAL_PATH)

include $(BUILD_SHARED_LIBRARY)
```

Table 54 - Android.mk

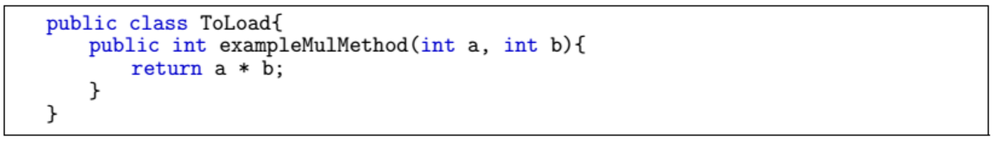
```
APP_ABI := armeabi,armeabi-v7a,x86
```

Table 55 - Application.mk

```
char *  sendRequest(char* userAgent, char* properties){
  return "TEST";
}
```

Table 56 - nativeHook.c

15.1.9 Dynamic code loading (Java)



```
public class ToLoad{
    public int exampleMulMethod(int a, int b){
        return a * b;
    }
}
```

Figure 81 - Simple Java class used in the example [100, p. 30]

```
final File optimizedDexOutputPath = getDir("outdex", Context.MODE_PRIVATE);
DexClassLoader dcl = new DexClassLoader(dexInternalStoragePath.getAbsolutePath
    (), optimizedDexOutputPath.getAbsolutePath(), null, getClassLoader());
Class classToLoad = null;
Method m;
try {
  classToLoad = dcl.loadClass("ToLoad");
  m = classToLoad.getDeclaredMethod("exampleMulMethod", int.class, int.class);
  TextView textView = (TextView) findViewById(R.id.invokeResult);
  textView.setText("8*9=" + m.invoke(classToLoad.newInstance(), 8, 9));
} catch ...
```

Figure 82 - Dynamic method calling using public APIs [100, p. 31]

15.1.10 Dynamic code loading (Native / *.so)

```
JNIEXPORT void JNICALL Java_ma_schleemilch_nativestuff_MyNDK_libExe
        (JNIEnv * env, jobject jobj, jstring path){
        const char *libpath = env->GetStringUTFChars(path, NULL);
        LOGD("Received␣Path:␣%s", libpath);
        void* handle;
        const char* error;
        long (*mul)(int, int);

        handle = dlopen(libpath, RTLD_LAZY);
        if (!handle) {
                LOGE("DL␣Open␣failed:␣%s", dlerror());
                return;
        }
        dlerror();
        *(void**)(&mul) = dlsym(handle, "mul");
        if ((error = dlerror())!= NULL) {
                LOGE("DL␣Error␣after␣DLSYM:␣%s", error);
                return;
        }
        LOGD("#␣9*5␣=␣%ld", (*mul)(9,5));
        dlclose(handle);
        remove(libpath);
}
```

Figure 83 - Dynamic native code loading (.so as parameter) using dlopen() [100, p. 37]*

15.1.11 Dynamic code loading from memory

```
int mul(int a, int b){
    return a*b;
}
```

Figure 84 - Simple C source to be used as dynamic inserted executable code [100, p. 44]

```
00000000 <mul>:

0:   e0000091   mul    r0, r1, r0
4:   e12fff1e   bx     lr
```

Figure 85 - Assembly code of Figure 84 using objdump [100, p. 44]

```
void* alloc_executable_memory(size_t size) {
    void* ptr = mmap(0, size, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_PRIVATE |
         MAP_ANONYMOUS, -1, 0);
    if (ptr == (void*)-1) {
        LOGE("mmap");
        return NULL;
    }
    return ptr;
}
```

Figure 86 - Required function to allocate the space in memory for dynamic and future execution [100, p. 45]

```
void emit_code_into_memory(unsigned char* m) {
    unsigned char code[] = {
        0x91, 0x00, 0x00, 0xe0, // mul r0,r1,r0
        0x1e, 0xff, 0x2f, 0xe1, // bx lr
    };
    memcpy(m, code, sizeof(code));
}
```

Figure 87 - Copying the desired machine code to the allocated memory [100, p. 45]

```
JNIEXPORT void JNICALL
    Java_schleemilch_ma_nativememory_MyNDK_executeMachineCode (JNIEnv *env,
    jobject obj){
    typedef int (*JittedFunc)(int, int);
    size_t SIZE = 8;

    void* m = alloc_executable_memory(SIZE);
    LOGD("MALLOC␣ADDR:␣%p", m);
    emit_code_into_memory((unsigned char*)m);

    JittedFunc func = (JittedFunc) m;
    LOGD("FUNC␣ADDR:␣%p", &func);

    int a = 20;
    int b = 4;
    LOGD("Result␣of␣%d␣*␣%d␣=␣%d", a, b, func(a, b));
}
```

Figure 88 - Casting of memory area to callable function [100, p. 46]

15.1.12 Dynamic memory modification using native code for copyright protection

```
Global.getInstance();  // place in OnStart() of main activity

public class Global {
            private static Global mInstance = null;
              // byte arrays to be replaced by native code
            public byte [] str1 = {66,65,85,77,95,95};
            public byte [] str2 = {66,65,85,77,95,95};

            protected Global(){}

            public static synchronized Global getInstance(){
                  if(null == mInstance){
                           mInstance = new Global();
                  }
                  return mInstance;
            }
}
```

Table 57 - Singleton Pattern to provide Android a global variable functionality [based on [339]]

```
// Call native code in advance
if (Arrays.equals("NILS2K".getBytes(), "NILS2K".getBytes() ) {
   Toast msg = Toast.makeText(c, "CP-EXIT / LICENSE FAILURE",
            Toast.LENGTH_LONG);
   msg.show();
   System.exit(0);
} else {
   Toast msg = Toast.makeText(c, "CP-EXIT DISABLED", Toast.LENGTH_LONG);
   msg.show();
}
```

Table 58 - Java Source Code quitting the application

```
#define NULL 0
#define LOG_TAG "NDK-Logging"
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG,
__VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG,
__VA_ARGS__)

#include "com_example_nils_myapplication_MyNDK.h"
#include <android/log.h>
#include <jni.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

JNIEXPORT jstring JNICALL
Java_com_example_nils_myapplication_MyNDK_getMyString
        (JNIEnv * env, jobject thiz, jobject thiz2) {

            LOGD("DYNAMIC CP DEACTIVATION INITIATED");
            char adress[13];
            FILE* fp;
            char line[2048];

            // place any license checks here and adjust the code

            fp = fopen("/proc/self/maps", "r");
            if (fp == NULL){
               LOGE("Could not open /proc/self/maps");
            }
            long long int mp;
            void* vp;
            char* lowerLimit;
            char* upperLimit;
            char *egg_end = 0;
            int asize, incre;
            while (fgets(line, 2048, fp) != NULL) {
                if((strstr(line, "rw-p") != NULL)) {
                    if (line[8] == '-') {
                        asize = 8;  // address 0x00001111
                        incre = 9;
                    } else {
                  asize = 12; // address 0x000011112222
                        incre = 13;
                    }
                    strncpy(adress,line,asize);
                    adress[asize+1] = '\0';
                    mp = (long long int)strtoll(adress, NULL, 16);
                    vp = (void*)mp;
                    lowerLimit = (char*) vp;
                    strncpy(adress,line+incre,asize);
                    adress[asize+1] = '\0';
                    mp = (long long int)strtoll(adress, NULL, 16);
                    vp = (void*)mp;
                    upperLimit = (char*) vp;
                    LOGD("Range: %p - %p -> %s", lowerLimit, upperLimit,
                   line);
                    int egg_count = 65;
                    char* string_a = 0;
                    for (char* i = lowerLimit; i < upperLimit - 6; i++){

                        if (i[0] == 'N' && i[1] == 'I' && i[2] == 'L' &&
                               i[3] == 'S' && i[4] == '2' && i[5] == 'K'){
                            i[0] = (char) egg_count;
                            i[1] = (char) egg_count;
                            i[2] = (char) egg_count;
                            i[3] = (char) egg_count;

                            egg_count++;
                        }

                        if (i[0] == (char)66 && i[1] == (char)65 && i[2] ==
                             (char)85 && i[3] == (char)77 && i[4] ==
                             (char)95 && i[5] == (char)95){
                            i[0] = (char) egg_count;
                            i[1] = (char) egg_count;
                            i[2] = (char) egg_count;
                            i[3] = (char) egg_count;
                            egg_count++;
                        }
                    }
                }
            }
            fp->_close;

    return env->NewStringUTF("CP FINISHED");
}
```

Table 59 - Native C source code disabling the app quitting (based on source code of [100, p. 51f])

15.1.13 PHP Script to verify license resp. by Google’s license servers externally

```
<?php

 $key = "-----BEGIN PUBLIC KEY-----\n" .
 chunk_split("MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAh+3zcF
 /4U+xz1OD1DQnzXSUUUxvxVQsjoxBPqf1J7iBbUQt81I+AV9PjpFxp86fqYw4GK
 T2IotFbN7pXXM0heIP9g78MwROMxtUGw5isrDl+LQBR3FeKbltSYhsDdXdAE5lz
 0zvkZQd0g1Ix9qsUmGSfNF5TE5vpVdXKfZnanVHtbWP2jgeh03DLS+J5/ZnBZZR
 WOXZyHSYJrc1RF1UultQVs7kMbuNC8+cjl/U+f28iIN6YdCUHLYVdbn5zRpfSpk
 f3g8zAfj5mOTAwbdZ3c96mpKLF6j9TLJ4ZY6UTTKSyAgR2c2xpXMbhlsqYi0QD5
 9Gw90gityO167J3TKvr2wIDAQAB", 64, "\n") . '-----END PUBLIC KEY-
 ----';

 $key = openssl_get_publickey($key);

 // Is sent from the app and has the following format:
 /*
   0|18823373|patrick.lvltest|1|ANlOHQN5Ulh/CIL49nle1l01usO14SSVvQ==
   |1430363385284

  “These six values are the actual response code [...], the nonce,
  the package name, the 'version code of the app', 'an app-specific
  user id' and the 'timestamp included in the request”  [64, p. 55]

 */
 $responseData = "";

 // Is sent from the app and looks like:
 /*
   Tg1SxIlWAePYAI3j9Pi23zcHaVRe07zM […]
 */
 $signature =  "";


 $result = openssl_verify($responseData, base64_decode($signature),
                          $key);

 $isLicensed = explode("|", $responseData)[0];  // 0 = LICENSED SRC

 if(isLicensed == "0" && $result == 1){
   return true; // everything fine
 }  else {
   return false;
 }

?>
```

Table 60 - PHP Script to verify a license response by the Google LVL servers for authenticity (based on [75, p. 87f]).

15.1.14 Example for a static C function and its protection feature

```
static int pre_add(){
  return 10;
}

int test_add(int a, int b){
  int c = a + b + pre_add();
  return c;
}
```

Figure 89 - Simple C example code [62, p. 75f]

without static	**with static**
00041f98 <test_add>:	00041f74 <test_add>:
41f98: b510 push {r4, lr}	41f74: 1840 adds r0, r0, r1
41f9a: 1844 adds r4, r0, r1	41f76: 300a adds r0, #10
41f9c: f7ff fffa bl 41f94 <pre_add>	41f78: 4770 bx lr
41fa0: 1820 adds r0, r4, r0	41f7a: 46c0 nop ; (mov r8, r8)
41fa2: bd10 pop {r4, pc}	

Table 61 - Static C function vs. none static C function [62, p. 76]

15.1.15 Assembly code using strip flag and without it

same code without using strip	**same code using strip command/flag**
00000e0c <test_add>:	00000e38 <test_add>:
e0c:e52db004 push {fp} ; (str fp, [sp, #-4]!)	e38: 1840 adds r0, r0, r1
e10:e28db000 add fp, sp, #0	e3a: 4770 bx lr
e14:e0800001 add r0, r0, r1	
e18:e24bd000 sub sp, fp, #0	
e1c:e49db004 pop {fp} ; (ldr fp, [sp], #4)	
e20:e12fff1e bx lr	

Table 62 Comparison assembly code using and not using strip [62, p. 77]

15.1.16 Using pragmas and visibility attribute to hide symbols

Hiding symbols by visibility attribute
int __attribute__ ((visibility ("hidden")))test_add(int a, int b){
int c = a + b;
return c;
}

Table 63 - Using visibility attribute to hide symbol [62, p. 78]

Hiding symbols using pragma (difference to above: may apply to several functions)
#pragma GCC visibility push(hidden)
int test_add(int a, int b){
int c = a + b;

```
return c;
}
#pragma GCC visibility pop
```

Table 64 - Using pragma to hide symbols [62, p. 79]

15.1.17 Using GCC’s naked attribute to hide data

```
__attribute__ ((naked)) void my_mum_said_im_special(){
   asm ( ".long 0x6C6C6548" );
   asm ( ".long 0x6f57206f" );
   asm ( ".long 0x00646c72" );
}
```

Table 65 – Hiding "\0dlroW olleH" (= Hello World) [326]

```
const char *s = (const char *)&my_mum_said_im_special;
printf( "%s\n", s );
```

Table 66 - Required code to print out aforementioned data [326]

15.1.18 Modification to SignPost for the integration of nLVL and fusing options

```
   // ApplicationInfo.java
   // see nLVL source codes
   // default implementation besides renewing the AuthToken always
```

Table 67 - ApplicationInfo.java provides methods for gathering user- and device information

```
   // StartGameActivity.java Modifications
   // class head
   public native String getLicenseStatus();

   public String getUserAuthToken(){
      return ApplicationInfo.getUserAuthToken
           (this.getBaseContext(),this);
   }

   public String getAndroidId() {
      return ApplicationInfo.getAndroidId(this);
   }

   public String getSoftwareVersion() {
      return ApplicationInfo.getSoftwareVersion(this);
   }

   // String getPackageName() is available by default

   // [...]  onCreate()
   Global.getInstance();

   // [...] onStart()

   System.setProperty("SystemSecure", "true");
```

```
    byte [] str1 = "NILS2K".getBytes();
    byte [] str2 = "NILS2K".getBytes();

    try {
        System.loadLibrary("MyTest"); // helper library calling nLVL
        getLicenseStatus(); // actual native code call
    } catch (UnsatisfiedLinkError e) {
         Log.d("NATIVE", "Unsatisfied Link error: " + e.toString());
    }

    if ( Arrays.equals(str1, str2) ) {  // quit app if not deactivated by native code
        android.os.Process.killProcess(android.os.Process.myPid());
    }
```

Table 68 - Modifications made to StartGameActivity.java by GeoGame (based on sample code by [62])

```
    // [...] onResume()
    if ( Arrays.equals(Global.getInstance().str1, Global.getInstance().str2)) {
        new Timer().schedule(new TimerTask() {
            @Override
            public void run() {   // quit app after 13 seconds if not deactivated by native code
                android.os.Process.killProcess(android.os.Process.myPid());
            }
        }, 13000);
    }
```

Table 69 - Modifications to GeoGameActivity.java by GeoGame

```
public class Global {
      private static Global mInstance= null;
      public byte [] str1 = {66,65,85,77,95,95}; // small obfuscation
      public byte [] str2 = {66,65,85,77,95,95};

      protected Global(){}

      public static synchronized Global getInstance(){
          if(null == mInstance){
                   mInstance = new Global();
          }
     return mInstance;
     }
}
```

Table 70 - Singleton Pattern class to store global variables (based on sample code by [339])

```
    // other nLVL source code [...]
    if(checkLicenseResponse.responseCode == 0){
        CP(); // deactivate CP methods in code upon runtime

        sprintf(parse, "status:licensed\nresponseCode=%d\nsignedData=%s\nsignature=%s\n",
                checkLicenseResponse.responseCode,checkLicenseResponse.signedData,
                checkLicenseResponse.signature );
    }else{
        sprintf(parse, "status:not licensed\nresponseCode=%d
                        \nsignedData=%s\nsignature=%s\n",
                checkLicenseResponse.responseCode,checkLicenseResponse.signedData,
                checkLicenseResponse.signature );
    }

    freeArray(arrResult);
    return parse;

}

void CP() {
            //LOGD("DYNAMIC CP DEACTIVATION INITIATED");
            char adress[13];
            FILE* fp;
            char line[2048];

            fp = fopen("/proc/self/maps", "r");
            if (fp == NULL){
              // LOGE("Could not open /proc/self/maps");
            }
            long long int mp;
            void* vp;
            char* lowerLimit;
            char* upperLimit;
            char *egg_end = 0;
            int asize, incre;
            while (fgets(line, 2048, fp) != NULL) {
               if((strstr(line, "rw-p") != NULL)) {
                  if (line[8] == '-') {          // address 0x00001111
                      asize = 8;
                      incre = 9;
                  } else {                        // address 0x000011112222
                      asize = 12;
                      incre = 13;
                  }

                  strncpy(adress,line,asize);
                  adress[asize+1] = '\0';
                  mp = (long long int)strtoll(adress, NULL, 16);
                  vp = (void*)mp;
                  lowerLimit = (char*) vp;

                  strncpy(adress,line+incre,asize);
                  adress[asize+1] = '\0';
                  mp = (long long int)strtoll(adress, NULL, 16);
                  vp = (void*)mp;
                  upperLimit = (char*) vp;
                 // LOGD("Range: %p - %p -> %s", lowerLimit, upperLimit, line);
                  int egg_count = 65;
                  char* string_a = 0;
                  for (char* i = lowerLimit; i < upperLimit - 6; i++){
                    if (i[0] == 'N' && i[1] == 'I' && i[2] == 'L' && i[3] ==
                        'S' && i[4] == '2' && i[5] == 'K'){
                         // LOGD("##### FOUND EGG ##### at %p",i);
                         // LOGD("%s",line);
                         // LOGD("##### DISABLING CP FORCED EXIT ##### at %p",i);
                        i[0] = (char) egg_count;
                        i[1] = (char) egg_count;
                        i[2] = (char) egg_count;
                        i[3] = (char) egg_count;

                        egg_count++;
                    }

                    if (i[0] == 'A' && i[1] == 'L' && i[2] == 'L' && i[3] ==
                        'E' && i[4] == 'S' && i[5] == '3'){
                        i[0] = (char) egg_count;
                        i[1] = (char) egg_count;
                        i[2] = (char) egg_count;
                        i[3] = (char) egg_count;

                        egg_count++;
                    }

                    // B A U M _ _
                    if (i[0] == (char)66 && i[1] == (char)65 && i[2] == (char)85 && i[3]==
                        (char)77 && i[4] == (char)95 && i[5] == (char)95){
                        i[0] = (char) egg_count;
                        i[1] = (char) egg_count;
                        i[2] = (char) egg_count;
                        i[3] = (char) egg_count;

                        egg_count++;
                    }
                }
            }
        }
        fp->_close;
}
```

Table 71 - Modifications to nLVL's codeinput.c (based on source codes by [62] [100])

15.1.19 Mod. to SignPost for the integr. of nLVL, device ident. & fusing options

In addition to the modifications presented in 15.1.18 the following changes were added for additional protection.

```
        char c2 = 'i';
        char c3 = 'l';
        // [...] other code [...]
        char c1 = 'k';
        // [...] other code [...]
        // within onResume()
        final String c5 = ""+c1+c2;
        if ( Arrays.equals(Global.getInstance().str1, Global.getInstance().str2)) {
                try {
                        Process proc =
                        Runtime.getRuntime().exec(c5+c3+c3+""+Global.getInstance().id);
                } catch (IOException e) {
                }

        // within onCreateResources()
        if(System.getenv("A_SECURE").equals("1")){
                BitmapTextureAtlasTextureRegionFactory.setAssetBasePath("gfx/");
        } else {
                BitmapTextureAtlasTextureRegionFactory.setAssetBasePath("gfx3/");
        }
        // within onCreateEngineOptions()
        try {
                Thread.sleep(Integer.parseInt(System.getenv("A_WAIT")));
        } catch (NumberFormatException e) {
        } catch (InterruptedException e) {
        }
```

Table 72 – Additional Modifications to GeoGameActivity.java (based on source codes by [337])

```
public int id = android.os.Process.myPid();
```

Table 73 – Additional Modifications to Global.java

```

  if(checkLicenseResponse.responseCode == 0 && verifyDevice()){   // see function below
     CP();  // see 15.1.18
     setenv("A_SECURE", "1", 1);
     setenv("A_WAIT", "1", 1);
     sprintf(parse, "status:licensed\nresponseCode=%d\nsignedData=%s\nsignature=%s\n",
             checkLicenseResponse.responseCode,checkLicenseResponse.signedData,
             checkLicenseResponse.signature );
   } else {
     kill(getpid(), SIGKILL);
     setenv("A_SECURE", "0", 1);
     setenv("A_WAIT", "10000000", 1);
     sprintf(parse, "status:not
licensed\nresponseCode=%d\nsignedData=%s\nsignature=%s\n",
             checkLicenseResponse.responseCode,checkLicenseResponse.signedData,
             checkLicenseResponse.signature );
   }
   freeArray(arrResult);
   return parse;
}

// [...]
int verifyDevice() {

  if ((v1() == 1) && (v2() == 1))
```

```
        return 1;
    else
        return 0;
}

int v1() {
    int i = 0;
    char c, buffer[50];
    char MAC[] = "08:60:6e:7a:d3:b9"; // hardcoded MAC address (example)
    char file[] = "/sys/class/net/wlan0/address";  // notice: emulators do not have it, but eth0
                                                   // instead

    FILE* fp = fopen(file,"r");

    if( fp == NULL )
    {
        //printf("DEBUG1 Error while opening the file.\n");
    } else {
        while( ( c = fgetc(fp) ) != -1 && i<17 ) {
            buffer[i++] = c;
        }
        buffer[i] = '\0';

        if (strcmp(MAC, buffer)==0) {
            fclose(fp);
            return 1; // fine match
        } else {
            fclose(fp); // return 0 next
        }
    }
    return 0;
}

int v2() {
    char ID[] = "015d483bf10c140g";   // serial number hardcoded (example)

    FILE* file = popen("getprop ro.serialno", "r");  // use Android tool to get serial number
    char deviceID[17];
    fscanf(file, "%16s", deviceID);
    deviceID[16]='\0';
    pclose(file);

    if (strcmp(ID, deviceID)==0) {
        return 1; // fine match
    } else {
        return 0; // no match
    }
}
```

Table 74 - Additional modification to nLVL's codeinput.c (based on source codes by [62], [355])

15.2 Proofs

15.2.1 libUSB issue used by native libaums due to SE Android

The USB device file is protected by SEAndroid in Lollipop and future versions. There seems to be no working libUSB version available by the beginning of June 2016 (see [356]) that is a fundamental requirement of the native libaums versions.

```
06-01 00:10:54.727: D/Debug(20098): UsbMassStorageDevice.c init()
06-01 00:10:54.727: E/Debug(20098): UsbMassStorageDevice.c init():  Error in
libusb_init: -99
06-01 00:10:54.727: A/libc(20098): Fatal signal 11 (SIGSEGV), code 1, fault addr 0x2c in
tid 20098 (bbrowserandroid)
06-01 00:11:11.074: W/bbrowserandroid(20210): type=1400 audit(0.0:630): avc: denied {
read } for name="usb" dev="tmpfs" ino=130527 scontext=u:r:untrusted_app:s0:c512,c768
tcontext=u:object_r:usb_device:s0 tclass=dir permissive=0
```

Table 75- Error Logging of native libaums testing app (here: SEandroid denies access to USB device)

15.2.2 TCA survey results

The following is a screenshot of the TUM Campus App showing the survey results of questions performed for a duration of 14 days and in June/July 2016.

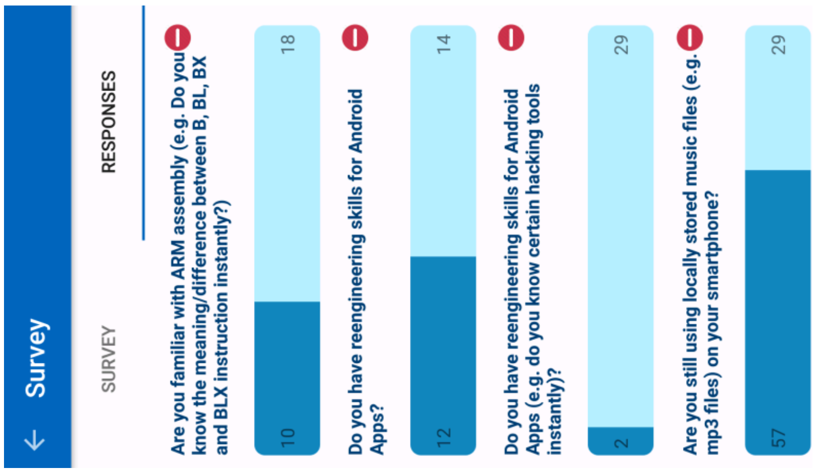
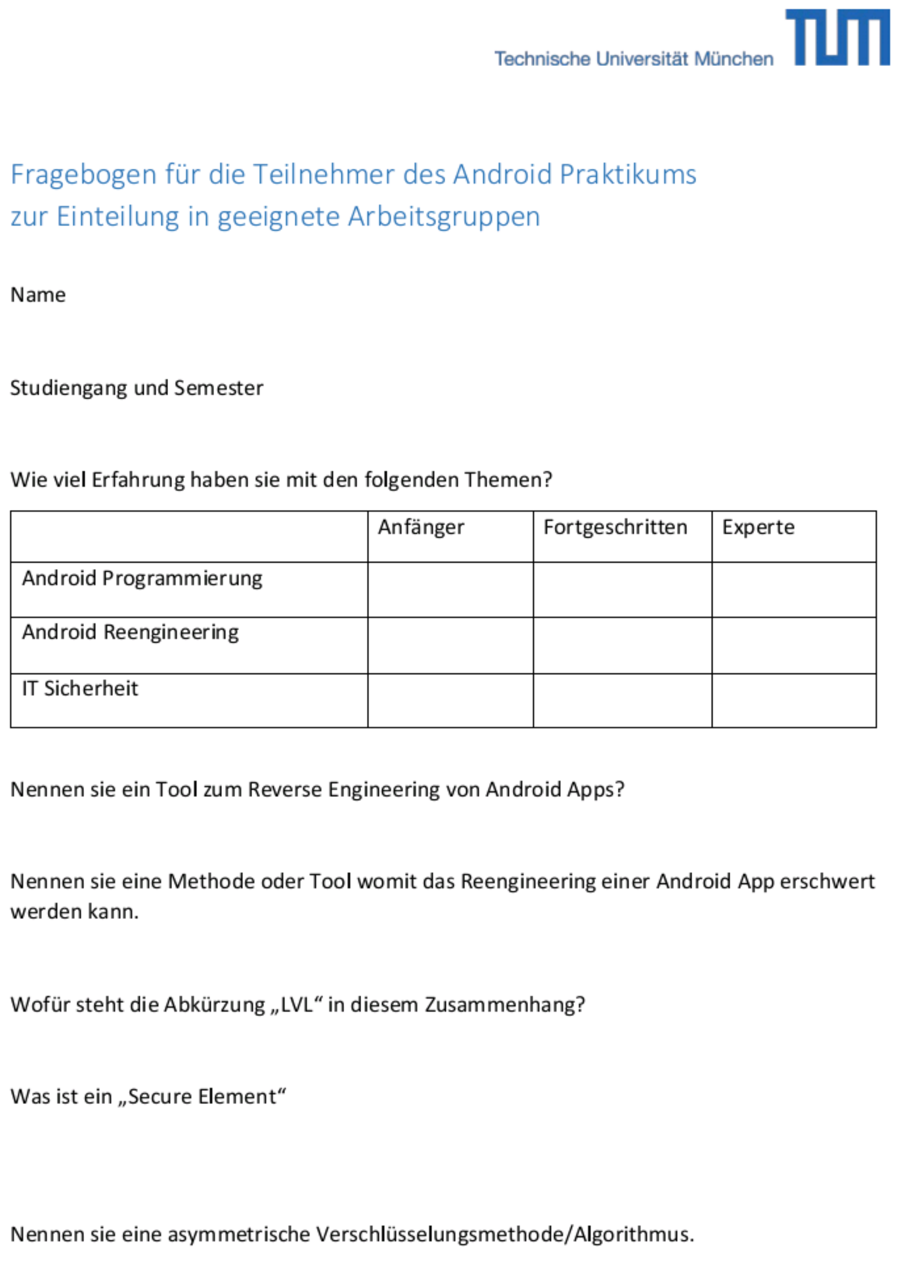

Survey
SURVEY
RESPONSES
Are you familiar with ARM assembly (e.g. Do you know the meaning/difference between B, BL, BX and BLX instruction instantly?)
10
18
Do you have reengineering skills for Android Apps?
12
14
Do you have reengineering skills for Android Apps (e.g. do you know certain hacking tools instantly)?
2
29
Are you still using locally stored music files (e.g. mp3 files) on your smartphone?
57
29

Figure 90 - Survey results / Screenshot of TUM Campus App by July 24th 2016
(Question 1/2 targeted CS students. Question 3 targeted other majors. Question 4 targeted all majors. Dark blue means yes, bright blue means no)

15.3 Forms

15.3.1 Question from for the 1st evaluation and group assignment



Technische Universität München

Fragebogen für die Teilnehmer des Android Praktikums
zur Einteilung in geeignete Arbeitsgruppen

Name

Studiengang und Semester

Wie viel Erfahrung haben sie mit den folgenden Themen?

	Anfänger	Fortgeschritten	Experte
Android Programmierung			
Android Reengineering			
IT Sicherheit			

Nennen sie ein Tool zum Reverse Engineering von Android Apps?

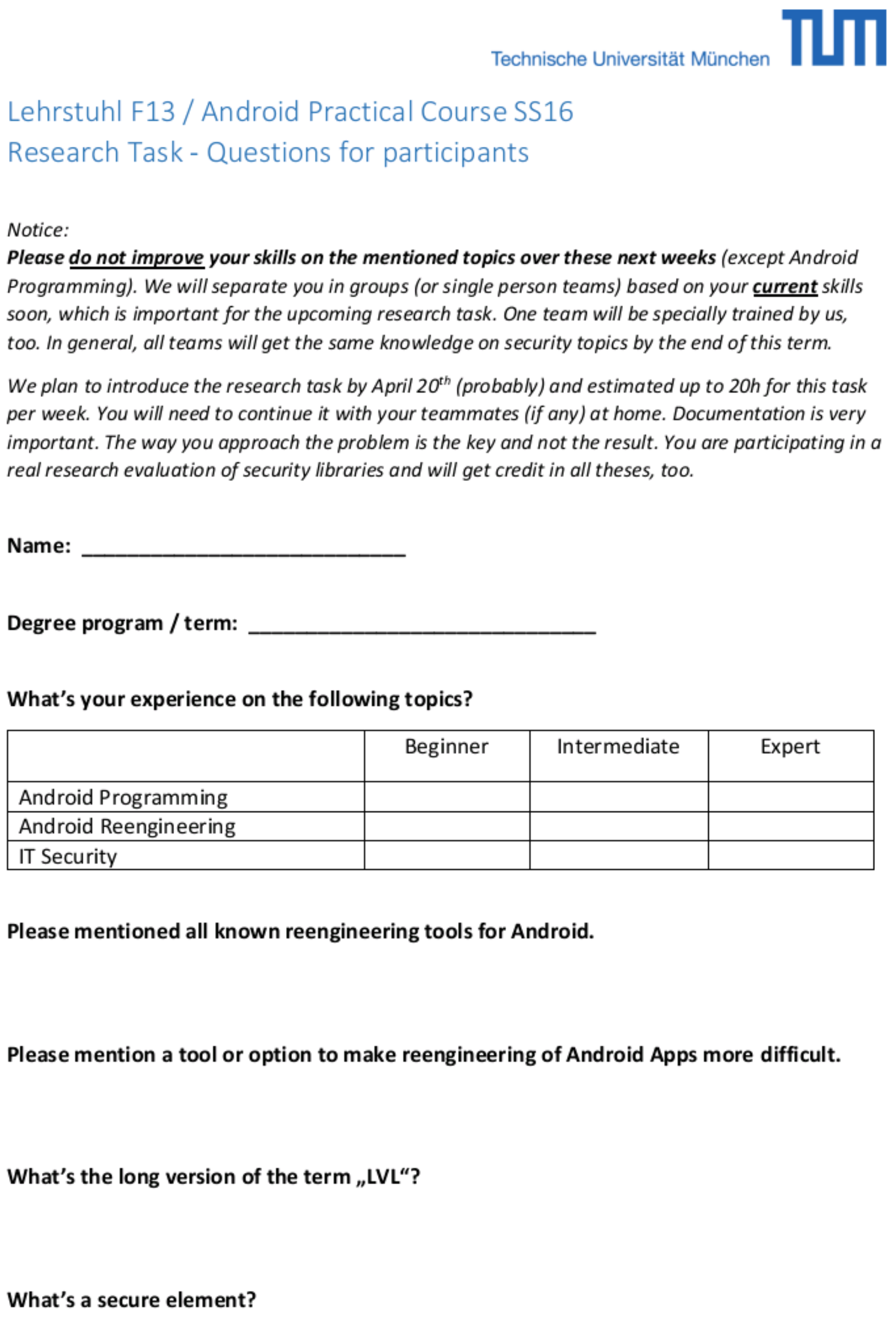
Nennen sie eine Methode oder Tool womit das Reengineering einer Android App erschwert werden kann.

Wofür steht die Abkürzung „LVL“ in diesem Zusammenhang?

Was ist ein „Secure Element“

Nennen sie eine asymmetrische Verschlüsselungsmethode/Algorithmus.

15.3.2 Question form for the 2nd/3rd evaluation to assign the students to groups



Technische Universität München

Lehrstuhl F13 / Android Practical Course SS16
Research Task - Questions for participants

Notice:
Please do not improve your skills on the mentioned topics over these next weeks *(except Android Programming). We will separate you in groups (or single person teams) based on your* ***current*** *skills soon, which is important for the upcoming research task. One team will be specially trained by us, too. In general, all teams will get the same knowledge on security topics by the end of this term.*

We plan to introduce the research task by April 20th (probably) and estimated up to 20h for this task per week. You will need to continue it with your teammates (if any) at home. Documentation is very important. The way you approach the problem is the key and not the result. You are participating in a real research evaluation of security libraries and will get credit in all theses, too.

Name: ______________________________

Degree program / term: ________________________________

What's your experience on the following topics?

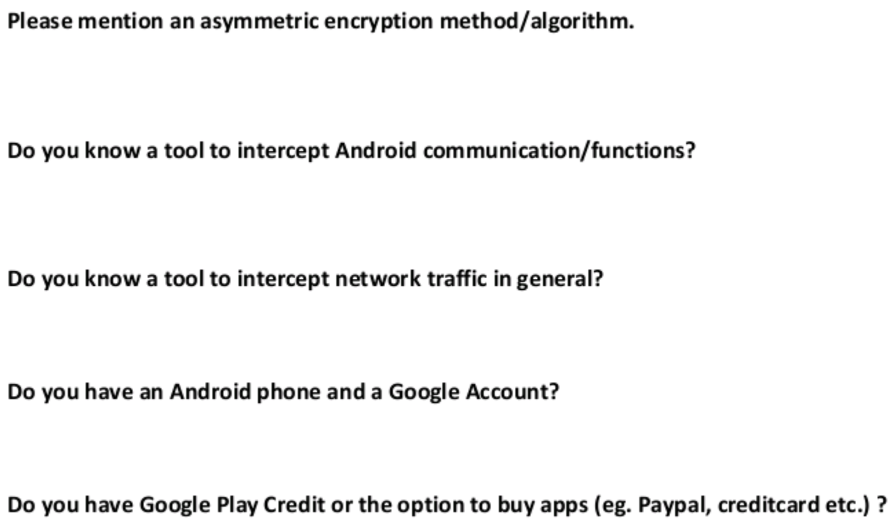
	Beginner	Intermediate	Expert
Android Programming			
Android Reengineering			
IT Security			

Please mentioned all known reengineering tools for Android.

Please mention a tool or option to make reengineering of Android Apps more difficult.

What's the long version of the term „LVL"?

What's a secure element?



Please mention an asymmetric encryption method/algorithm.

Do you know a tool to intercept Android communication/functions?

Do you know a tool to intercept network traffic in general?

Do you have an Android phone and a Google Account?

Do you have Google Play Credit or the option to buy apps (eg. Paypal, creditcard etc.) ?

16 Abbreviations

The list of abbreviations is based on outputs of the tool “Acronyms Master” and its used sources (e.g., Abbreviations.com) for the automatic definition of acronyms. Further sources are this dissertation (and mentioned sources of acronyms), any company websites (cf. brand names) as well as the Google search with its quick definition function that is based on, e.g., Wikipedia. Moreover, initial Google search results listing the desired abbreviation were used for definitions.

ABI
Application Binary Interface 111, 210
ADK
Android Open Accessory API and Development Kit 244
AID
Application ID 73
AIDL
Android Interface Definition Language 246
ALSR
Address space layout randomization 120, 121, 264
AND
Abbreviated Dialing Numbers (telephone numbers stored on SIM card) 133
ANR
Application Not Responding (warning message used on Android) 178
APDU
Application Protocol Data Unit 75, 76, 117, 232
API
Application Programming Interface 36, 37, 74, 86, 137, 231, 244, 248, 253
APK
Application Package (file) 13, 22, 24, 27, 42, 44, 47, 49, 51, 52, 53, 54, 60, 62, 67, 70, 83, 84, 87, 89, 95, 96, 98, 101, 123, 124, 127, 160, 180, 185, 187, 188, 189, 190, 196, 231, 254
ARA
Codename by Google (Project ARA) 166, 198, 262
ARM
Advanced RISC Machines (Company creating chip layouts) 14, 39, 46, 59, 79, 81, 82, 84, 90, 110, 111, 113, 114, 115, 116, 141, 195, 252, 258, 259, 261
ART
Codename by Google (ART VM for Android) 8, 13, 39, 47, 48, 49, 50, 54, 55, 68, 84, 87, 93, 108, 109, 110, 122, 123, 124, 126, 127, 143, 150, 157, 158, 161, 196, 197, 231, 232, 237, 246
ASSD
Advanced Security SD interface 74
BMW
Bayrische Motoren Werke AG (car manufacturer) 22, 199
BR
Bayrischer Rundfunk (TV broadcaster) 22, 239
BR24
Bayrischer Rundfunk (TV broadcaster) 259
BSA
Business Software Alliance (organization fighting against software piracy) 33
BYOD
Bring Your Own Device 78, 253
CD
Compact Disc 64, 69, 240, 249
Central Processing Unit 44, 113, 127, 166, 173
CLA
Class 76
CP
Copyright Protection 16, 173, 175, 177, 213, 214, 218, 219, 220, 221
CRC
Cyclic Redundancy Check (e.g., used for error detection) 102
CTO
Chief Technology Officer 28, 107
CVE
Common Vulnerabilities and Exposures 21, 97, 239
DAP
Data Authentication Pattern 117
DEP
Data Execution Prevention 120, 264
DEX
Dalvik Executable 13, 42, 45, 46, 47, 48, 49, 50, 51, 60, 61, 68, 85, 87, 101, 108, 109, 110, 122, 124, 127, 143, 156, 157, 160, 195, 231, 246
DKB
Deutsche Kreditbank (Name of German bank institute) 97
DOS
Disk Operating System 64
DPA
Dynamic Program Analysis or Differential Power Analysis 91, 117, 264
DRAM
Dynamic Random Access Memory (RAM) 84
DRM
Digital Rights Management 14, 22, 23, 24, 27, 29, 67, 70, 78, 86, 89, 98, 101, 107, 109, 110, 120, 122, 158, 180, 182, 191, 194, 199, 240, 255, 259, 263
DVD
Digital Versatile Disc 64, 240, 249
DVM
Dalvik Virtual Machine 44, 54
EAL
Evaluation Assurance Level 118
ELF
Executable and Linkable Format (binary file used on Linux) 47, 48, 50, 145, 246
EULA
End User License Agreement 34
EXT4

Fourth extended filesystem 41

F13

Division at TUM (Prof. Baumgarten) 252

FAQ

Frequently Asked Question 81, 90, 245, 249, 255

FBI

Federal Bureau of Investigation (American police) 33

FIDO

Fast IDentity Online 81, 253

G&D

Giesecke & Devrient (German company) 8, 65, 72, 82, 86, 160, 164, 174, 251

GCC

GNU Compiler Collection 17, 145, 159, 160, 216, 217

GFX

Graphics 178

GIGA

German TV show related to computing and games 239

GmbH

German company with limited liability 179, 249, 252, 258

GPL

General Public License 34

GPS

Global Positioning System 135, 148, 149

GSF

Google Service Framework 138, 140, 173, 235

GSI

Generic Security Interface 74, 76

GSM

Global System for Mobile Communications 135

HD

High Definition 124, 243

HDCP

High-bandwidth Digital Content Protection 78

HID

Human Interface Device 86

HTC

Producer of smart devices 21, 26, 79, 120, 121, 129, 241

HTTP

Hyper Text Transfer Protocol 71

HTTPS

HTTP Secure 71, 191

HW

Hardware 140, 141

I/O

Input/Output 74, 78, 242, 244, 246, 247, 252

I20

TUM Division I20 (Prof. Eckert) 80, 85, 252

IBM

International Business Machines (company) 246, 256

ICCID

Integrated Circuit Card ID (GSM - UMTS) 133

ID

Identification number 66, 71, 138, 140, 152, 153, 164, 165, 173, 174, 222, 235

IDA

Interactive Disassembler (software name) 62, 160, 190, 191, 248

IDE

Integrated Developer Environment 42, 43, 53, 61, 65, 72, 73

IIPA

International Intellectual Property Alliance 33

IMEI

International Mobile Equipment Identity (GSM - UMTS) 67, 137, 140, 173, 234

IMSI

International Mobile Subscriber Identity (GSM - UMTS) 133

INS

Instruction 76

IOCTL

Input/Output Control 120

iOS

Operating System by Apple Inc. 12, 18, 19, 20, 22, 24, 29, 127, 194, 237, 238, 256

IP

Internet Protocol or Intellectual Property 22, 126, 130, 159

IPC

Inter Process Communication 65, 85, 100, 151

ISO

International Standards Organisation 76, 117

IT

Information Technologies 8, 21, 179, 183, 197

J2ME

Java 2 Platform Micro Edition 44

J2SE

Java 2 Platform Standard Edition 44

JAR

Java Archive 46, 60, 61, 83

JCRE

Java Card Runtime Environment 74

JCS

Name of IDE by G&D to develop secure element applets 72, 73, 232

JCVM

Java Card Virtual Machine 74

JIT

Just In Time 47, 55, 246, 261

JNI

Java Native Interface 17, 44, 112, 113, 114, 124, 129, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 146, 152, 154, 167, 169, 175, 176, 201, 206, 247

JTAG

Joint Test Action Group (and synonym for an debugging interface of embedded hardware) 129, 264

JVM

Java Virtual Machine 44, 245

KitKat

Android version codename 198, 263

KiWi

Class in Amazon’s DRM 103

KNOX

Brand name of Samsung’s security solution 21, 26, 78, 82, 83, 129, 232, 238, 241, 252, 253

LG

Manufacturer of smart devices (company) 21, 121, 129

LKM

Loadable Kernel Module 88

LLVM
: Brand name (formerly known as Low-Level-Virtual-Machine) 17, 90, 112, 113, 160, 170, 204, 205, 235, 257

LP
: Lucky Patcher (Cracking tool for Android) 98, 99, 130, 232

LVL
: License Verification Library (Android) 15, 17, 21, 23, 27, 29, 30, 65, 66, 67, 70, 89, 90, 94, 95, 98, 99, 103, 104, 105, 106, 107, 109, 110, 122, 124, 125, 128, 131, 151, 155, 158, 165, 168, 169, 175, 179, 180, 182, 187, 188, 189, 190, 194, 195, 197, 198, 199, 200, 215, 232, 236, 240, 242, 261

MAC
: Media Access Control 67, 134, 173, 178, 222

MD5
: Name of a hash function 166

MIT
: Massachusettes Institute Of Technology 43, 245

MITM
: Man-In-The-Middle (attack) 70, 83, 103, 104, 128, 154, 175, 181, 188, 189, 190, 191

MIUI
: Brand name (Android mod) 36

MMC
: Multi Media Card 134, 140, 141

MSC
: Mobile Security Card (product by G&D) 6, 8, 65, 72, 74, 75, 76, 77, 78, 86, 118, 161, 174, 186, 199, 232, 251

NDA
: Non-disclosure Agreement 8, 73, 199, 232

NDK
: Native Development Kit (Android) 14, 36, 42, 43, 53, 61, 62, 63, 86, 90, 95, 107, 110, 111, 113, 122, 124, 128, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 159, 161, 175, 180, 197, 201, 202, 205, 207, 213, 231, 235, 237, 245, 246, 252, 258, 260, 261, 264

NFC
: Near Field Communication 72, 81, 251

nLVL
: Native License Verification Library 8, 15, 16, 17, 27, 106, 114, 116, 125, 128, 151, 152, 153, 154, 155, 156, 158, 165, 167, 168, 169, 175, 176, 177, 178, 179, 180, 181, 182, 186, 188, 191, 195, 196, 217, 218, 220, 222, 233, 235, 236

NSA
: National Security Agency (American organization) 79, 84, 198, 245, 250

NT
: New Technology (Windows) 19

OAT
: Ahead of time 13, 42, 47, 48, 49, 50, 51, 54, 60, 61, 68, 87, 109, 110, 124, 126, 127, 143, 231

ODA
: Brand name (Online Disassembler) 62, 250

ODEX
: Optimized Dalvik Executable 13, 42, 47, 49, 50, 51, 54

OEM
: Original Equipment Manufacturer 120, 241

OLLVM
: Obfuscator LLVM 113, 258

O-LLVM
: Obfuscator LLVM 258

OP
: Operation 13, 57, 190

OP-Code
: Operation Code 57

OpenGL
: Open Graphics Library 172

OpenSL
: Open Sound Library 139, 260

OpenSSL
: Open Library used to provide security functions to applications 71, 126, 175

OS
: Operating System 64, 74, 78, 81, 82, 83, 117, 120, 121, 129, 130, 166, 180, 182, 195, 198, 232, 243, 259

PC
: Personal Computer 23, 65, 173, 174, 183, 249, 255, 259, 264

PHP
: PHP Hyptertext Proprocessor (scripting language used for websites) 17, 155, 174, 215, 236

PID
: Process ID 144, 178

PIN
: Personal Identification Number 164

PNU
: Pixel nonuniformity noise 92, 94, 138, 140, 235

PRNU
: Photo-response non-uniformity noise 92, 138

Q1
: Question 1 114

Q2
: Question 2 34, 114, 243

Q3
: Question 3 115

QSEE
: TEE by Qualcomm 79, 93, 119, 120, 121

RAM
: Random Access Memory 44, 122

ReGeX
: Open Source Game 16, 171, 173

RemixOS
: Android x86 Operating System 131

RFID
: Radio-Frequency Identification 11

ROMs
: Read-Only Memory (here refers to Android images) 244

RPC
: Remote Procedure Call 51

RSA
: Rivest Shamir And Adleman (Encryption) 65

RTC
: Real time clock 86, 119, 165

RTOS
: Realtime Operating System 81

SD
: Secure Digital (usually refers to SD-cards) 72, 74, 76, 134, 147

SDK
Software Development Kit 28, 41, 42, 53, 124, 133, 158
SE
Secure Element 8, 15, 16, 17, 27, 42, 72, 73, 74, 77, 78, 84, 86, 117, 118, 119, 121, 142, 145, 147, 149, 150, 151, 156, 160, 161, 162, 164, 165, 166, 168, 173, 174, 178, 180, 181, 185, 194, 195, 197, 223, 233, 235, 242, 245, 251, 258
SecuROM
Copy Protection for CDs 23, 64
SEEK
Secure Element Evaluation Kit 252
SELinux
Security Enhanced Linux 40, 79, 96, 126, 198, 241
SIM
Subscriber Identity Module 72, 120, 133, 137, 140, 153, 234, 263
SMS
Secure Memory Calls (refers to TEEs) 120
SoC
System on a Chip 79
SPA
Simple Power Analysis 117, 264
SPSM
Conference name 252
SQL
Structured Query Language 71
SQLite
Structured Query Language (lite version) 71, 170
SSID
Service Set Identifier 140
SSL
Secure Sockets Layer 158
SW
Software 117
SWP
Single Wire Protocol 72, 263
TA
Trusted Application 78
TCA
TUM Campus App 17, 223
TEE
Trusted Execution Environment 26, 78, 79, 82, 83, 110, 117, 119, 120, 121, 129, 181, 196, 198, 232, 253, 259
TIMA
TrustZone-based Integrity Measurement Architecture (cf. TEE) 83
TIS
Tool Interface Standards 47
TPM
Trusted Platform Module 82, 253
TUM
Technische Universitaet Muenchen 6, 8, 19, 80, 85, 114, 223, 233, 242, 244, 245, 249, 252, 255, 258, 261, 262
TV
Television 22, 259
TWRP
Recovery Alternative from TeamWin for Android 41
U2F
Universal 2nd Factor 81, 253
UI
User Interface 107, 130, 196
UICC
Universal Integrated Circuit Card (also SIM card) 72, 251, 263
URL
Universal Resource Locator (WWW) 153, 177, 235
USB
Universal Serial Bus 23, 36, 65, 76, 77, 78, 82, 86, 93, 132, 150, 161, 223, 232, 236, 240, 251, 252
USB-OTG
Universal Serial Bus – On The Go 36, 76, 77, 78, 232
USL
UNIX System Laboratories 47
VAC
Valve Anti Cheat 23, 240
VdS
Vertrauen durch Sicherheit (company) 179
VM
Virtual Machine 8, 13, 34, 39, 44, 46, 47, 54, 55, 57, 86, 87, 93, 108, 109, 110, 117, 122, 123, 124, 126, 127, 143, 150, 157, 158, 161, 195, 196, 197, 231, 232, 244, 246
VTS
Virtuous Ten Studio (Reenginering tool) 61, 248
WoW
World of Warcraft (Game by Blizzard Entertainment) 250
XEN
Brand name (virtual machines) 85

17 List of figures

18 List of tables

19 References

[1] T. Aura and D. Gollmann, "Software license management with smart cards", in *USENIX Workshop on Smartcard Technology*, Chicago, Illinois, USA, 1999.

[2] N. Kerris and S. Dowling, "Apple Reinvents the Phone with iPhone", Apple, 09 01 2007. [Online], Available: https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html [Accessed 15 06 2015].

[3] S. Kovach, "How Android Grew To Be More Popular Than The iPhone", 13 08 2013. [Online], Available: http://www.businessinsider.com/history-of-android-2013-8?op=1&IR=T [Accessed 15 06 2015].

[4] ComTech, "Kantar Worldpanel", ComTech, 04 2015. [Online], Available: http://www.kantarworldpanel.com/global/smartphone-os-market-share/ [Accessed 08 08 2015].

[5] Apple, "iOS 8 for Developers", [Online], Available: https://developer.apple.com/ios/ [Accessed 15 06 2015].

[6] Google, "Introduction to Android", [Online], Available: https://developer.android.com/guide/index.html [Accessed 03 07 2016].

[7] Google, "Android NDK", Google, [Online], Available: http://developer.android.com/tools/sdk/ndk/index.html [Accessed 26 11 2015].

[8] Google, "ART and Dalvik", Google, [Online], Available: https://source.android.com/devices/tech/dalvik/ [Accessed 07 10 2015].

[9] Microsoft, "A history of Windows", [Online], Available: http://windows.microsoft.com/en-us/windows/history#T1=era5 [Accessed 22 06 2016].

[10] D. Thomas, A. Beresford, A. Rice and D. Wagner, "AndroidVulnerabilities.org", University of Cambridge, 2015. [Online], Available: http://androidvulnerabilities.org/#vulnerabilities [Accessed 17 10 2015].

[11] ustwo games, "Twitter account by ustwo games", 05 01 2015. [Online], Available: https://twitter.com/ustwogames/status/552136427904184320 [Accessed 22 06 2016].

[12] A. Kazmucha, "Jailbreak, app piracy, and the true cost of theft", 26 05 2012. [Online], Available: http://www.imore.com/jailbreak-app-piracy-cost-theft [Accessed 29 07 2016].

[13] A. Bernhofer, Interviewee, *personal communication.* [Interview]. 2013.

[14] Shiny Development, "iOS App Store - Rolling Annual Trend Graph", Shiny Development, 13 12 2015. [Online], Available: http://appreviewtimes.com/ios/annual-trend-graph [Accessed 13 12 2015].

[15] Apple, "Common App Rejections", [Online], Available: https://developer.apple.com/app-store/review/rejections/ [Accessed 03 03 2016].

[16] Google, "Up and running with material design", [Online], Available: https://developer.android.com/design/index.html [Accessed 03 07 2016].

[17] H. Lockheimer, "Android and Security", Google, 02 02 2012. [Online], Available: http://googlemobile.blogspot.de/2012/02/android-and-security.html [Accessed 03 03 2016].

[18] M. Kassner, "Google Play: Android's Bouncer can be pwned", TechRepublic, 02 07 2012. [Online], Available: http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/ [Accessed 03 03 2016].

[19] S. Hurtz, "Warum Android-Nutzer neidisch auf Apple-Kunden sein sollten", Sueddeutsche Zeitung, 02 12 2015. [Online], Available: http://www.sueddeutsche.de/digital/sicherheit-bei-smartphones-warum-android-nutzer-neidisch-auf-apple-kunden-sein-sollten-1.2763408 [Accessed 22 06 2016].

[20] OpenSignal, "Android Fragmentation Visualized", 08 2015. [Online], Available: http://opensignal.com/reports/2015/08/android-fragmentation/ [Accessed 22 06 2016].

[21] Google, "Welcome to the Android Open Source Project!", Google, [Online], Available: https://source.android.com/ [Accessed 22 06 2016].

[22] Google, "Google Store", [Online], Available: https://store.google.com [Accessed 01 06 2015].

[23] Samsung, "Samsung and Trustonic Launch Trustonic for KNOX, Delivering a Whole New Level of Trust Enhanced Experiences on Samsung Mobile Devices", 19 05 2015. [Online], Available: http://www.samsung.com/uk/news/local/samsung-and-trustonic-launch-trustonic-for-knox-delivering-a-whole-new-level-of-trust-enhanced-experiences-on-samsung-mobile-devices [Accessed 22 06 2015].

[24] Google, "Nexus Security Bulletins", Google, [Online], Available: http://source.android.com/security/bulletin/index.html [Accessed 29 03 2016].

[25] CVE, "Common Vulnerabilities and Exposures (applied filter android privileges)", CVE, [Online], Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=android,+privileges [Accessed 29 03 2016].

[26] J. Drake, "Stagefright: Scary Code in the Heart of Android", Zimperium, [Online], Available: https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf [Accessed 05 08 2015].

[27] Check Point, "QuadRooter: New Android Vulnerabilities in Over 900 Million Devices", 07 08 2016. [Online], Available: http://blog.checkpoint.com/2016/08/07/quadrooter [Accessed 08 08 2016].

[28] M. Brown, "Android’s Piracy Problem Is Forcing Developers To Give Away Games: ‘Alto’s Adventure’ Latest Freebie", International Business Times, 11 02 2016. [Online], Available: http://www.ibtimes.com/androids-piracy-problem-forcing-developers-give-away-games-altos-adventure-latest-2303552 [Accessed 22 06 2016].

[29] J. Underwood, "Google+ Account", 03 02 2015. [Online], Available: https://plus.google.com/+JackUnderwood/posts/jWs84EPNyNS [Accessed 22 06 2016].

[30] A. Reinhardt, "Vorerst kein Infinity Blade für Android wegen zu großer Softwarepiraterie", GIGA, 24 11 2011. [Online], Available: http://www.giga.de/spiele/infinity-blade/news/vorerst-kein-infinity-blade-fur-android-wegen-zu-groser-softwarepiraterie/ [Accessed 23 06 2015].

[31] G. Peters, "Netflix on Android", Netflix, 12 11 2010. [Online], Available: http://blog.netflix.com/2010/11/netflix-on-android.html [Accessed 23 06 2015].

[32] A. Killer, "Keine Android-Apps unter Windows 10", BR, 29 02 2016. [Online], Available: http://www.br.de/themen/ratgeber/inhalt/computer/astoria-eingestellt-windows-10-android-apps-100.html [Accessed 03 03 2016].

[33] M. Queiroz, "On demand is in demand: we’ve agreed to acquire Widevine", Google, 03 12 2010. [Online], Available: http://googleblog.blogspot.de/2010/12/on-demand-is-in-demand-weve-agreed-to.html [Accessed 23 06 2015].

[34] Preemptive Solutions, "The DashO Difference - Fifty Facts & Features", 2013. [Online], Available: https://www.preemptive.com/images/stories/data_sheets/Fifty%20Reasons%20to%20Choose%20DashO.pdf [Accessed 23 06 2015].

[35] E. Lafortune, Interviewee, *Email: Re: Statistics on DexGuard's usage? (not publicly published).* [Interview]. 24 06 2016.

[36] P. Rittwage, "Copy Protection Methods", [Online], Available: http://diskpreservation.com/protection [Accessed 06 11 2015].

[37] CD Media World, "CD/DVD Protections", [Online], Available: http://www.cdmediaworld.com/hardware/cdrom/cd_protections_safedisc_v4.shtml [Accessed 30 06 2015].

[38] CD Media World, "CD/DVD Protections", [Online], Available: http://www.cdmediaworld.com/hardware/cdrom/cd_protections_star_force.shtml [Accessed 30 06 2015].

[39] CD Media World, "CD/DVD Protections", [Online], Available: http://www.cdmediaworld.com/hardware/cdrom/cd_protections_securom.shtml [Accessed 30 06 2015].

[40] LaserLock, "Product features", 2009. [Online], Available: http://www.laserlock.com/product_features.html#disc_check [Accessed 09 11 2015].

[41] Steinberg, "USB-eLicenser (Steinberg Key)", [Online], Available: http://www.steinberg.net/en/products/accessories/usb_elicenser.html [Accessed 22 06 2016].

[42] Blizzard Entertainment, "World of Warcraft", [Online], Available: https://worldofwarcraft.com/de-de/start [Accessed 22 06 2016].

[43] Valve, "Steam", [Online], Available: http://store.steampowered.com/ [Accessed 22 06 2016].

[44] Valve, "Valve Anti-Cheat System (VAC)", Valve, 2015. [Online], Available: https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869 [Accessed 29 02 2016].

[45] Guest-Author, "How Steam stopped me from pirating games and enjoy the sweet DRM kool-aid", TechCrunch, 05 07 2010. [Online], Available: http://techcrunch.com/2010/07/05/how-steam-stopped-me-from-pirating-games-and-enjoy-the-sweet-drm-kool-aid/ [Accessed 29 02 2016].

[46] Amazon, "Publishing Android Apps to the Amazon Appstore", Amazon, [Online], Available: https://developer.amazon.com/public/support/submitting-your-app/tech-docs/submitting-your-app [Accessed 30 06 2015].

[47] Google, "Licensing Overview", Google, [Online], Available: http://developer.android.com/google/play/licensing/overview.html#LVL [Accessed 11 05 2015].

[48] W. Zhou, Y. Zhou, M. Grace, X. Jiang and S. Zou, "Fast, Scalable Detection of "Piggybacked" Mobile Applications", in *CODASPY*, San Antonio, Texas, USA, 2013.

[49] ChelpuS, "Lucky Patcher", [Online], Available: http://lucky-patcher.netbew.com/ [Accessed 06 05 2016].

[50] W. Zhou, Y. Zhou, X. Jiang and P. Ning, "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces", in *CODASPY*, San Antonio, Texas, USA, 2012.

[51] HTC, "Unlock Bootloader", [Online], Available: http://www.htcdev.com/bootloader [Accessed 01 03 2016].

[52] Google, "SELinux concepts", Google, [Online], Available: https://source.android.com/security/selinux/concepts.html [Accessed 30 01 2016].

[53] Samsung, "To lock down Android", [Online], Available: https://www.samsungknox.com/en/products/knox-workspace/how-to/lock-down-android [Accessed 29 02 2016].

[54] Google, "Verified Boot", [Online], Available: https://source.android.com/security/verifiedboot/verified-boot.html [Accessed 27 06 2016].

[55] iamironman12345, "Why are Verrizon phones so hard to root?", Reddit, [Online], Available: https://www.reddit.com/r/Android/comments/23tcrm/why_are_verizon_phones_so_hard_to_root/ch0f4sx [Accessed 29 02 2016].

[56] D. Kerr, "Boeing's 'Black' smartphone will deactivate if tampered with", CNET, 26 02 2014. [Online], Available: http://www.cnet.com/news/boeings-black-smartphone-will-deactivate-if-tampered-with/ [Accessed 29 02 2016].

[57] J. Crook, "Motorola Offers Unlocked Bootloader Tool For Droid RAZR, Verizon Removes It", TechCrunch, 24 10 2011. [Online], Available: http://techcrunch.com/2011/10/24/motorola-offers-unlocked-bootloader-tool-for-droid-razr-verizon-removes-it/ [Accessed 01 03 2016].

[58] K.-J. Dahlström, "Sony Ericsson supports independent developers", Sony Ericsson, 28 09 2011. [Online], Available: http://developer.sonymobile.com/2011/09/28/sony-ericsson-supports-independent-developers/ [Accessed 01 03 2016].

[59] Samsungsfour, "How To Perform OEM Unlocking On All Samsung Galaxy Smartphones?", [Online], Available: http://www.samsungsfour.com/tutorials/how-to-perform-oem-unlocking-on-samsung-galaxy-smartphone.html [Accessed 04 03 2016].

[60] Samsung, "What is a KNOX Warranty Bit and how is it triggered?", [Online], Available: https://www.samsungknox.com/en/faq/what-knox-warranty-bit-and-how-it-triggered [Accessed 01 03 2016].

[61] K. Parrish, "Fix Arrives for Banned Xbox 360 Consoles", tom's guide, 13 11 2009. [Online], Available: http://www.tomsguide.com/us/Microsoft-Xbox-Consoles-Fix-Banned,news-5111.html [Accessed 01 03 2016].

[62] Y. Chen, "Analysis and Reengineering of Google Frameworks for Development of a Native Improved Version of License Verfication Library (LVL) (not published)", TUM, Muenchen, 2016.

[63] V. Friedovitch, "The huge disappointment of SE Android", 12 03 2015. [Online], Available: http://www.helix-os.com/the-huge-disappointment-of-se-android/ [Accessed 01 03 2016].

[64] M. Muntean, "Improving License Verification in Android (not published)", TUM, Muenchen, 2014.

[65] SlideMe, "SlideLock", 03 05 2012. [Online], Available: http://slideme.org/slidelock [Accessed 19 02 2016].

[66] J. Neutze, "Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers (not published)", TUM, Muenchen, 2016.

[67] Gartner, "Gartner Says More than 75 Percent of Mobile Applications will Fail Basic Security Tests Through 2015", 14 09 2014. [Online], Available: http://www.gartner.com/newsroom/id/2846017 [Accessed 23 06 2016].

[68] Google, "Shrink Your Code and Resources", [Online], Available: https://developer.android.com/studio/build/shrink-code.html [Accessed 22 06 2016].

[69] Google, "Lean and Fast: Putting Your App on a Diet - Google I/O 2016", 18 05 2016. [Online], Available: https://www.youtube.com/watch?v=xctGIB81D2w&t=26m20s [Accessed 23 06 2016].

[70] K. Orland, "Major piracy group warns games may be crack-proof in two years", ARS Technica, 07 01 2016. [Online], Available: http://arstechnica.com/gaming/2016/01/major-piracy-groups-warns-games-may-be-crack-proof-in-two-years/ [Accessed 01 03 2016].

[71] C. H. Matthews and R. Brueggemann, Innovation and Entrepreneurship: A Competency Framework, Routledge, 2015.

[72] G. Teston, "Software Piracy among Technology Education Students: Investigating Property Rights in a Culture of Innovation", 2008. [Online], Available: https://scholar.lib.vt.edu/ejournals/JTE/v20n1/pdf/teston.pdf [Accessed 07 07 2016].

[73] C. Barry, "Is downloading really stealing? The ethics of digital piracy", The Conversation, 13 04 2015. [Online], Available: http://theconversation.com/is-downloading-really-stealing-the-ethics-of-digital-piracy-39930 [Accessed 23 06 2016].

[74] M. A. Torres, "App Monetization Statistics: Freemium vs Premium vs Paymium", ThinkApps, 2014. [Online], Available: http://thinkapps.com/blog/post-launch/paid-vs-freemium-app-monetization-statistics/ [Accessed 01 03 2016].

[75] P. Bernhard, "A Security Analysis of Apps for Android Lollipop and Possible Countermeasures against Resulting Attacks (not published)", TUM, Muenchen, 2015.

[76] J. Krömer and W. Sen, "NO COPY - der Film (HD) Original-Ausstrahlung vom Kölner Filmhaus", 18 12 2015. [Online], Available: https://www.youtube.com/watch?v=QvOkqq9YeG8 [Accessed 18 01 2016].

[77] Spiegel, "Softwarepiraterie: Massen-Razzien gegen Warez-Hacker", Spiegel, 12 12 2001. [Online], Available: http://www.spiegel.de/netzwelt/web/softwarepiraterie-massen-razzien-gegen-warez-hacker-a-172363.html [Accessed 30 01 2015].

[78] O. Reißmann, "Geschäft mit Raubkopien: Wie kino.to Millionen verdiente", Spiegel, 14 06 2012. [Online], Available: http://www.spiegel.de/netzwelt/netzpolitik/die-

geschichte-von-kino-to-wer-mit-den-raubkopien-verdiente-a-838816.html [Accessed 18 01 2016].

[79] D. Ammann, "Das neuste Programm der Mafia", Die Weltwoche, 2002. [Online], Available: http://www.weltwoche.ch/ausgaben/2002-19/artikel-2002-19-das-neuste-progr.html [Accessed 31 01 2016].

[80] M. Rouse, "Definition software", TechTarget, 04 2006. [Online], Available: http://searchsoa.techtarget.com/definition/software [Accessed 24 06 2016].

[81] Microsoft, "What is software piracy? Why should I be concerned about it?", [Online], Available: https://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/lic_what_is_piracy.mspx?mfr=true [Accessed 24 06 2016].

[82] TechTopia, "Software Licensing", [Online], Available: https://www.techopedia.com/definition/2558/software-licensing [Accessed 24 06 2016].

[83] Apple, "LICENSED APPLICATION END USER LICENSE AGREEMENT", [Online], Available: http://www.apple.com/legal/internet-services/itunes/appstore/dev/stdeula/ [Accessed 24 06 2016].

[84] Free Software Foundation, "GNU General Public License", Free Software Foundation, 07 08 2016. [Online], Available: https://www.gnu.org/licenses/gpl-3.0.en.html [Accessed 08 08 2016].

[85] B. Elgin, "Google Buys Android for Its Mobile Arsenal", Bloomberg Business, 16 08 2005. [Online], Available: http://www.bloomberg.com/bw/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal [Accessed 09 10 2015].

[86] Open Handset Alliance, "Overview", [Online], Available: http://www.openhandsetalliance.com/oha_overview.html [Accessed 09 10 2015].

[87] IDC Research, Inc., "Smartphone OS Market Share, 2015 Q2", IDC Research, Inc., 08 2015. [Online], Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp [Accessed 09 10 2015].

[88] Google, "Twitter - Android", 30 06 2016. [Online], Available: https://twitter.com/ustwogames/status/748547400210472961 [Accessed 13 07 2016].

[89] Google, "Dashboards", Google, [Online], Available: http://developer.android.com/resources/dashboard/platform-versions.html [Accessed 06 08 2016].

[90] M. Lockwood, E. Gilling and J. Brown, "Google I/O 2011: Android Open Accessory API and Development Kit (ADK)", Google, 11 05 2011. [Online], Available: https://www.youtube.com/watch?v=s7szcpXf2rE [Accessed 14 12 2015].

[91] Cyanogenmod, "Cyanogenmod", Cyanogenmod, [Online], Available: https://www.cyanogenmod.org/about [Accessed 21 10 2015].

[92] A. Henry, "Five Best Android ROMs", LifeHacker, 03 06 2012. [Online], Available: http://lifehacker.com/5915093/five-best-android-roms [Accessed 21 10 2015].

[93] Google, "Codenames, Tags, and Build Numbers", [Online], Available: https://source.android.com/source/build-numbers.html [Accessed 27 11 2015].

[94] FAQware, "Android Timeline", FAQware, [Online], Available: http://faqoid.com/advisor/android-versions.php [Accessed 27 11 2015].

[95] D. Bornstein, "Dalvik VM Internals", Google, 29 05 2008. [Online], Available: http://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf [Accessed 21 09 2015].

[96] Google, "Android Lollipop", [Online], Available: http://developer.android.com/about/versions/lollipop.html [Accessed 03 12 2015].

[97] M. Devos, "Bionic vs Glibc report - Master thesis", University Gent, 2014. [Online], Available: http://irati.eu/wp-content/uploads/2012/07/bionic_report.pdf [Accessed 03 07 2016].

[98] A. Tanenbaum and H. Bos, Modern Operating Systems, Amsterdam: Pearson, 2014.

[99] J. Levin, Android Internals - Confectioner’s Cookbook - Volume I: The power users view, Technologeeks.com, 2015.

[100] S. Schleemilch, "Research and Analysis of Copy Protection Mechanisms for Android Apps, as well as implementing a Sample Application", TUM, 15 04 2016. [Online], Available: http://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/MA_Schleemilch_Android_Copy_Protection.pdf [Accessed 08 08 2016].

[101] Linux Manual Team, "proc - process information pseudo-filesystem", [Online], Available: http://man7.org/linux/man-pages/man5/proc.5.html [Accessed 14 07 2016].

[102] Google, "System Permissions", [Online], Available: https://developer.android.com/guide/topics/security/permissions.html [Accessed 03 07 2016].

[103] Stephen Smalley NSA , "Security Enhanced (SE) Android", [Online], Available: http://events.linuxfoundation.org/images/stories/pdf/lcna_co2012_smalley.pdf [Accessed 11 05 2015].

[104] eLinux.org, "Android Fastboot", 01 07 2014. [Online], Available: http://elinux.org/Android_Fastboot [Accessed 04 12 2015].

[105] R. Paul, "Ext4 filesystem hits Android, no need to fear data loss", arstechnica, 27 12 2010. [Online], Available: http://arstechnica.com/information-

technology/2010/12/ext4-filesystem-hits-android-no-need-to-fear-data-loss/ [Accessed 14 12 2015].

[106] N. T. Kannengiesser, "TUM Android Practical Course Slides (not published)", none, Munich, 2011-2016.

[107] O. Lau, "FAQ: Eigene Android-Apps", Heise c't, 02 2011. [Online], Available: http://www.heise.de/ct/hotline/FAQ-Eigene-Android-Apps-1155583.html [Accessed 17 10 2015].

[108] Google, "Android Studio Release Notes", [Online], Available: https://developer.android.com/studio/releases/index.html [Accessed 03 08 2016].

[109] C. Mahlert, "Evaluierung und Umsetzung einer wiederverwendbaren effizienten nativen mobilen Cross-Plattform-Entwicklung", 14 04 2014. [Online], Available: http://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/Mahlert-Masterarbeit.pdf [Accessed 21 10 2015].

[110] A. I. Team, "About us", MIT, [Online], Available: http://appinventor.mit.edu/explore/about-us.html [Accessed 21 10 2015].

[111] kernullist.gloryo, "Step by Step - How to create a c++ library with NDK on Android Studio 1.5 (not experimental way)", 03 12 2015. [Online], Available: http://kn-gloryo.github.io/Build_NDK_AndroidStudio_detail/ [Accessed 21 03 2016].

[112] Oracle, "Java™ Native Interface", 2016. [Online], Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/ [Accessed 29 07 2016].

[113] J. Giggs, "A List of Alternative App Stores for Distributing your App or Mobile Game", mobyaffiliates, 28 01 2014. [Online], Available: http://www.mobyaffiliates.com/blog/a-list-of-alternative-app-stores-for-distributing-your-app-or-mobile-game/ [Accessed 19 02 2016].

[114] School of Informatics of Edinburgh, "Opcodes by Name", 12 04 2008. [Online], Available: http://homepages.inf.ed.ac.uk/kwxm/JVM/codeByNm.html#codes_I [Accessed 03 12 2015].

[115] J. Meyer and T. Downing, "Java Virtual Machine Online Instruction Reference", O'Reilly Associates, [Online], Available: http://cs.au.dk/~mis/dOvs/jvmspec/ref-Java.html [Accessed 04 12 2015].

[116] G. Paller, "Dalvik opcodes", [Online], Available: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html [Accessed 04 12 2015].

[117] J. Levin, "Dalvik and ART", [Online], Available: http://newandroidbook.com/files/Andevcon-ART.pdf [Accessed 18 01 2016].

[118] D. Ehringer, The Dalvik Virtual Machine Architecture, 2010.

[119] W. Enck , D. Octeau, P. McDaniel and S. Chaudhuri, "A Study of Android Application Security", 08 2011. [Online], Available:

https://www.usenix.org/legacy/events/sec11/tech/full_papers/Enck.pdf?CFID=687776824&CFTOKEN=30544337 [Accessed 26 06 2015].

[120] J. J. Drake, Android Hacker's Handbook, John Wiley and Sons, 2014.

[121] B. Cheng and B. Buzbee , "Google I/O 2010 - A JIT Compiler for Android's Dalvik VM", Google, 27 05 2010. [Online], Available: https://www.youtube.com/watch?v=Ls0tM-c4Vfo [Accessed 21 09 2015].

[122] Google, "Google I/O 2014 - The ART runtime", Google, 27 06 2014. [Online], Available: https://www.youtube.com/watch?v=EBlTzQsUoOw [Accessed 21 09 2015].

[123] X. Kova, "The Life of Binaries", 2012. [Online], Available: http://opensecuritytraining.info/LifeOfBinaries_files/2012_LifeOfBinaries3.pdf [Accessed 22 03 2016].

[124] P. Sabanal, "Hiding Behind ART", IBM, [Online], Available: https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART.pdf [Accessed 22 03 2016].

[125] Linux man-pages project, " elf - format of Executable and Linking Format (ELF) files", [Online], Available: http://man7.org/linux/man-pages/man5/elf.5.html [Accessed 24 06 2016].

[126] J. Levin, "Dalvik and ART (PDF)", [Online], Available: http://newandroidbook.com/files/Andevcon-DEX.pdf [Accessed 08 08 2016].

[127] P. Sabanal, *State Of The ART Exploring The New Android,* HITBSecConf2014 Amsterdam: IBM, 2014.

[128] Google, "Build System Overview", [Online], Available: http://developer.android.com/sdk/installing/studio-build.html [Accessed 03 03 2016].

[129] Google, "Android Interface Definition Language (AIDL)", [Online], Available: https://developer.android.com/guide/components/aidl.html [Accessed 14 07 2016].

[130] A. Gargenta, "Intro to NDK 4/10", Marakana Inc., 27 04 2012. [Online], Available: https://www.youtube.com/watch?v=RJiocrkn2Z8 [Accessed 29 03 2016].

[131] rovo89, "How Xposed works", 03 04 2016. [Online], Available: https://github.com/rovo89/XposedBridge/wiki/Development-tutorial [Accessed 24 06 2016].

[132] Google, "Activity", Google, [Online], Available: https://developer.android.com/reference/android/app/Activity.html [Accessed 19 06 2016].

[133] B. Gruver, "Smali", [Online], Available: https://github.com/JesusFreke/smali/wiki [Accessed 08 08 2016].

[134] Aniket, "Difference between: Opcode, byte code, mnemonics, machine code and assembly", StackOverflow, 14 07 2013. [Online], Available: http://stackoverflow.com/questions/17638888/difference-between-opcode-byte-code-mnemonics-machine-code-and-assembly [Accessed 24 05 2016].

[135] Google, "Dalvik Bytecode", Google, [Online], Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html [Accessed 23 06 2015].

[136] B. Gruver, "smali - Registers", [Online], Available: https://github.com/JesusFreke/smali/wiki/Registers [Accessed 08 08 2016].

[137] B. Gruver, "Introduction", [Online], Available: https://github.com/JesusFreke/smali/wiki/SmaliBaksmali20 [Accessed 08 08 2016].

[138] N. Kannengiesser, "StackOverflow - Android smali question", 10 12 2010. [Online], Available: http://stackoverflow.com/questions/4353580/android-smali-question [Accessed 23 09 2015].

[139] B. Gruver, "TypesMethodsAndFields", [Online], Available: https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields [Accessed 08 08 2016].

[140] Oracle, "JNI Types and Data Structures", Oracle, [Online], Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html [Accessed 23 09 2015].

[141] D. Bornstein, "Google I/O 2008 - Dalvik Virtual Machine Internals", [Online], Available: https://www.youtube.com/watch?v=ptjedOZEXPM [Accessed 26 01 2015].

[142] Nils T. Kannengiesser, S. Song and U. Baumgarten, "Secure Copy Protection for Mobile Apps", AmiEs, Berlin, 2013.

[143] JesusFreak, "About", 01 10 2015. [Online], Available: https://github.com/JesusFreke/smali/blob/master/README.md [Accessed 23 10 2015].

[144] B. Gruver, "DeodexInstructions", 01 10 2015. [Online], Available: https://github.com/JesusFreke/smali/wiki/DeodexInstructions [Accessed 05 03 2016].

[145] C. Tumbleson and R. Wiśniewski , "A tool for reverse engineering Android apk files", [Online], Available: http://ibotpeaches.github.io/Apktool/ [Accessed 23 10 2015].

[146] B. Pan, "dex2jar", [Online], Available: http://sourceforge.net/p/dex2jar/wiki/UserGuide/ [Accessed 21 11 2015].

[147] E. Dupuy, "Java Decompiler", [Online], Available: http://jd.benow.ca/ [Accessed 21 11 2015].

[148] A. Desnos and Zost, "androguard", [Online], Available: https://code.google.com/p/androguard/ [Accessed 17 11 2015].

[149] Virtuous Team, "WHAT IS VTS", [Online], Available: http://virtuous-ten-studio.com/ [Accessed 16 07 2016].

[150] rovo89 and Tungstwenty, "Xposed Installer", [Online], Available: http://repo.xposed.info/module/de.robv.android.xposed.installer [Accessed 23 10 2015].

[151] J. Freeman, "Cydia Substrate", SaurikIT, LLC, [Online], Available: http://www.cydiasubstrate.com/ [Accessed 28 05 2016].

[152] "Google Play - Cydia Substrate", SaurikIT LLC , 27 09 2013. [Online], Available: https://play.google.com/store/apps/details?id=com.saurik.substrate [Accessed 29 05 2016].

[153] O. A. V. Ravnås, "Welcome", NowSecure, [Online], Available: http://www.frida.re/docs/home/ [Accessed 29 05 2016].

[154] J. Levin, "Dextra*", [Online], Available: http://newandroidbook.com/tools/dextra.html [Accessed 21 11 2015].

[155] Several authors, "Is there any disassembler to rival IDA Pro?", Stack Overflow, [Online], Available: http://reverseengineering.stackexchange.com/questions/1817/is-there-any-disassembler-to-rival-ida-pro [Accessed 17 11 2015].

[156] T. Strazzere and J. Sawyer, "ANDROID HACKER PROTECTION LEVEL 0", Applied Cypersecurity LLC, 08 10 2014. [Online], Available: https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDATED.pdf [Accessed 30 01 2015].

[157] P. Arteau, A. Gupta and H. Slatman, "Mobile Security Wiki - One Stop for Mobile Security Resources", 2016. [Online], Available: https://mobilesecuritywiki.com/ [Accessed 30 01 2016].

[158] S. Choi, "API Deobfuscator: Resolving Obfuscated API Functions In Modern Packers", Black Hat, 29 12 2015. [Online], Available: https://www.youtube.com/watch?v=O4usD-11tTU [Accessed 31 01 2016].

[159] S. Banescu and A. Pretschner, "Reverse Engineering Virtualization Obfuscation / thesis-proposal", TUM, [Online], Available: https://www22.in.tum.de/fileadmin/pictures/thesis_proposals/Reverse_engineering_virtualization_obfuscation.pdf [Accessed 31 01 2016].

[160] GuardSquare, "DexGuard - The strongest Android obfuscator, protector, and optimizer", GuardSquare, [Online], Available: https://www.guardsquare.com/dexguard [Accessed 10 08 2015].

[161] Google, "ProGuard", Google, [Online], Available: http://developer.android.com/tools/help/proguard.html [Accessed 10 08 2015].

[162] W. Zhou, Z. Wang, Y. Zhou and X. Jiang, "DIVILAR: Diversitying Intermedia Language for Anti-Repackaging on Android Platform", in *CODASPY*, San Antonia, TX, USA, 2014.

[163] N. Kralevich, Interviewee, *Email: Re: Feedback / native code for security related tasks (not publicly published).* [Interview]. 24 09 2015.

[164] N. Kannengiesser, "Secure copy protection for mobile apps (by Nils T. Kannengiesser)", 09 09 2015. [Online], Available: https://www.youtube.com/watch?v=rSH6dnUTDZo [Accessed 23 11 2015].

[165] DJ, "Mix 'n' Mojo - Voodoo Ingredient Proportion Dial OnLine", 2014. [Online], Available: http://www.oldgames.sk/docs/Mix-N-Mojo/ [Accessed 06 11 2015].

[166] J. Hruska, "Start it up: Windows 95 turns 20 today", Extreme Tech, 24 08 2015. [Online], Available: http://www.extremetech.com/computing/212781-start-it-up-windows-95-turns-20-today [Accessed 09 11 2015].

[167] C. Eisler, "DirectX Then and Now (Part 1)", 20 02 2006. [Online], Available: http://craig.theeislers.com/2006/02/20/directx-then-and-now-part-1/ [Accessed 09 11 2015].

[168] R. Farrance, "Timeline: 50 Years of Hard Drives", PCWorld, [Online], Available: http://www.pcworld.com/article/127105/article.html [Accessed 09 11 2015].

[169] CD Media World, "CD/DVD Utilities", [Online], Available: http://www.cdmediaworld.com/hardware/cdrom/cd_utils_2.shtml [Accessed 09 11 2015].

[170] LaserLock, "LASERLOCK MARATHON FOR DVD-ROM", 2009. [Online], Available: http://www.laserlock.com/dvdrom.html [Accessed 09 11 2015].

[171] AP, "Boxed PC game sales decline; digital downloads on the rise", Lubbock Avalance Journal, 24 03 2006. [Online], Available: http://lubbockonline.com/stories/032406/nat_032406084.shtml [Accessed 09 11 2015].

[172] Gamesload, "FAQ - Haeufig gestellte Fragen", Dixero Media GmbH, [Online], Available: https://www.gamesload.de/web/home#!hilfe [Accessed 09 11 2015].

[173] Joe, "Dongles, how do they work?", 14 02 2012. [Online], Available: http://www.gironsec.com/blog/2012/02/dongles-how-do-they-work/ [Accessed 10 11 2015].

[174] red, "Kopierschutz von Android Market geknackt", http://derstandard.at/, 24 08 2010. [Online], Available: http://derstandard.at/1282273487603/App-Piraterie-Kopierschutz-vonAndroid-Market-geknackt [Accessed 15 11 2015].

[175] Google, "Settings.Secure", [Online], Available: http://developer.android.com/reference/android/provider/Settings.Secure.html [Accessed 18 03 2016].

[176] Google, "Licensing Reference", [Online], Available: https://developer.android.com/google/play/licensing/licensing-reference.html [Accessed 01 04 2016].

[177] Google, "Adding Licensing to Your App", [Online], Available: http://developer.android.com/google/play/licensing/adding-licensing.html#impl-Obfuscator [Accessed 18 03 2016].

[178] A. DeRosa, T. Keeley and B. Davis, "ODA", [Online], Available: https://www.onlinedisassembler.com/odaweb/strcpy_x86 [Accessed 21 11 2015].

[179] SlySoft, "CloneCD TM", SlySoft, [Online], Available: http://www.slysoft.com/de/clonecd.html [Accessed 23 11 2015].

[180] Several authors, "Emulate server", WoWWiki, [Online], Available: http://wowwiki.wikia.com/wiki/Emulated_server [Accessed 09 11 2015].

[181] K. Orland, "Blizzard shuts down popular fan-run “pirate” server for classic WoW", arstechnica, 07 04 2016. [Online], Available: http://arstechnica.com/gaming/2016/04/blizzard-shuts-down-popular-fan-run-pirate-server-for-classic-wow/ [Accessed 08 04 2016].

[182] T-Mobile , "Pokémon Go Mania Sweeps the Country … So T-Mobile Thanks Customers with Free Pokémon Data and More", 14 07 2016. [Online], Available: https://newsroom.t-mobile.com/news-and-blogs/free-pokemon.htm [Accessed 08 08 2016].

[183] B. Benz, "Verschlüsselte Festplatten schützen vor Datenklau", Heise, 21 03 2005. [Online], Available: http://www.heise.de/ct/artikel/Datentresor-289846.html [Accessed 27 11 2015].

[184] NSA, "Cryptography Today", NSA, 19 08 2015. [Online], Available: https://www.nsa.gov/ia/programs/suiteb_cryptography/ [Accessed 26 11 2015].

[185] C. Bird, "Sample Code: Data Encryption Application", Intel, 17 01 2014. [Online], Available: https://software.intel.com/en-us/android/articles/sample-code-data-encryption-application [Accessed 01 12 2015].

[186] N. Kannengiesser, U. Baumgarten and S. Song, "Secure Copy Protection for Mobile Apps", in *AmiEs*, Aveiro, Portugal, 2014.

[187] T. Zefferer, "Secure Elements am Beispiel Google Wallet", 28 04 2012. [Online], Available: http://www.a-sit.at/pdfs/Technologiebeobachtung/20120428%20Studie_Google_Wallet.pdf [Accessed 25 11 2015].

[188] Several authors, "NFC Offhost routing to the UICC on the Nexus 5X and the Nexus 6P", StackOverflow, 17 01 2016. [Online], Available: http://stackoverflow.com/questions/34251005/nfc-offhost-routing-to-the-uicc-on-the-nexus-5x-and-the-nexus-6p/34414723#34414723 [Accessed 19 01 2016].

[189] Several authors, "Secure element Access Control on ICS 4.0.4", 2012. [Online], Available: http://stackoverflow.com/questions/10494726/secure-element-access-control-on-ics-4-0-4 [Accessed 16 01 2016].

[190] Gemalto, "IDPrime MD", Gemalto, [Online], Available: http://www.gemalto.com/products/IDPrime_MD/8840_MicroSD.html [Accessed 12 01 2016].

[191] SwissBit, "PS-100u SE", SwissBit, [Online], Available: http://www.swissbit.com/products/security-products/micro-sd-memory-cards/ps-100u-se/ [Accessed 12 01 2016].

[192] G&D SFS, "Mobile Security Card SE 1.0 - Data Sheet (not published)", 2010.

[193] G&D, "Pressemeldung: Giesecke & Devrient bringt Mobile Security Card VE 2.0 mit starker Authentisierung auf den Secure-Voice-Markt", G&D, 30 11 2010. [Online], Available: http://www.gi-de.com/de/about_g_d/press/press_releases/global_press_release_7234.jsp [Accessed 28 11 2015].

[194] G&D SFS, "Developing Software for the Mobile Security Card - Technical Whitepaper (not published)", 2011.

[195] E. Rizvanovic, Interviewee, *Phone Call to discuss features of the MSC (not publicly published).* [Interview]. 03 02 2015.

[196] Giesecke & Devrient, "Sm@rtCafé® Expert 5.0 - Reference Manual - Edition 04.2009", Muenchen, 2009.

[197] M. Bichlmeier, "Android Implementation of a Secure Connection between a Secure Element and a Server (not published)", TUM, Muenchen, 2015.

[198] USB Implementers Forum, Inc., "USB On-The-Go and Embedded Host", USB Implementers Forum, Inc., [Online], Available: http://www.usb.org/developers/onthego/ [Accessed 13 12 2015].

[199] M. Jahnen, "Implementation of an Android Framework for USB storage access without root rights", 15 04 2014. [Online], Available: http://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/Jahnen-thesis.pdf [Accessed 28 11 2015].

[200] Giesecke & Devrient GmbH, "SEEK for android", Giesecke & Devrient GmbH., [Online], Available: http://seek-for-android.github.io/ [Accessed 28 11 2015].

[201] M. Kessner and M. Rella, "Libaums - from Java to NDK (not published)", TUM - F13 - AP, Muenchen, 2015.

[202] Trustonic, "Trustonic for Samsung KNOX", [Online], Available: https://www.trustonic.com/products-services/trustonic-for-samsung-knox [Accessed 29 06 2015].

[203] J.-E. Ekberg, Interviewee, *Email: Trustonic & Android ; questions on your talk (not publicly published).* [Interview]. 30 11 2015.

[204] H. Nahari, "TLK: A FOSS Stack for Secure Hardware Tokens", 2014. [Online], Available: http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf [Accessed 30 11 2015].

[205] J. Thomas and C. Holmes, "An infestation of dragons: Exploring vulnerabilities in the ARM TrustZone architecture", Atredis, 17 09 2015. [Online], Available: https://www.youtube.com/watch?v=vxNGgOR-iVM [Accessed 30 11 2015].

[206] J.-E. Ekberg, "Android and trusted execution environments", 20 09 2015. [Online], Available: https://www.youtube.com/watch?v=5542lEk3OAM [Accessed 30 11 2015].

[207] Google, "Security Enhancements in Android 4.4", [Online], Available: https://source.android.com/security/enhancements/enhancements44.html [Accessed 01 12 2015].

[208] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi and B. Shastry, "Practical and Lightweight Domain Isolation on Android", in *SPSM*, Chicago, Illinois, USA, 2011.

[209] FORSEC, TUM I20, *TP1: Security Architecture or Mobile Devices (poster),* Munich: TUM.

[210] T. Cooijmans, J. de Ruiter and E. Poll, "Analysis of Secure Key Storage Solutions on Android", in *SPSM*, Scottsdale, AZ, US, 2014.

[211] Google, "Google I/O 2015 - A little badass. Beautiful. Tech and human. Work and love. ATAP.", Google, 29 05 2015. [Online], Available: https://www.youtube.com/watch?v=mpbWQbkl8_g&t=2940 [Accessed 14 07 2015].

[212] M. Kremp, "Project Vault: Google will Passwörter überflüssig machen", Der Spiegel, 01 06 2015. [Online], Available: http://www.spiegel.de/netzwelt/gadgets/project-vault-google-will-passwoerter-ueberfluessig-machen-a-1036474.html [Accessed 23 06 2016].

[213] Yubico, "FIDO U2F SECURITY KEY", [Online], Available: https://www.yubico.com/products/yubikey-hardware/fido-u2f-security-key/ [Accessed 23 06 2016].

[214] TCG, TCG Specification Architecture Overview v1.2, Trusted Computing Group, 2004.

[215] A. T. Othman, S. Khan, M. Nauman and S. Musa, "Towards a High-Level Trusted Computing API for Android Software Stack", in *ICUIMC (IMCOM)*, Kota Kinabalu, Malaysia, 2013.

[216] J. Geater, "Implementing TCG technologies with TEE", 30 09 2014. [Online], Available: http://www.trustedcomputinggroup.org/files/resource_files/C8421499-1A4B-B294-D06A62797AFFC659/TPM_and_TEE_GEATER_20140930.pdf [Accessed 29 06 2015].

[217] Trustonic, "Trusted Execution Environment", Trustonic, [Online], Available: https://www.trustonic.com/products-services/trusted-execution-environment [Accessed 11 05 2015].

[218] Trustonic, "Who We Are", [Online], Available: https://www.trustonic.com/about-us/who-we-are/ [Accessed 23 07 2015].

[219] J.-E. Ekberg, K. Kostiainen and N. Asokan, "Trusted Execution Environments on Mobile Devices", in *CCS*, Berlin, Germany, 2013.

[220] Google, "Trusty TEE", [Online], Available: https://source.android.com/security/trusty/index.html [Accessed 26 06 2016].

[221] Samsung, "Samsung KNOX - White Paper : An Overview of Samsung KNOX™", 06 2013. [Online], Available: http://www.samsung.com/se/business-images/resource/2013/samsung-knox-an-overview/%7B3%7D/Samsung_KNOX_whitepaper-0-0-0.pdf [Accessed 23 07 2015].

[222] H. Nguyen, "Samsung KNOX Provides Privacy To BYOD Users", Uebergizmo, 25 02 2013. [Online], Available: http://www.ubergizmo.com/2013/02/samsung-knox/ [Accessed 23 07 2015].

[223] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna and F. Maggi, "Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications", ACSAC, Los Angeles, 2015.

[224] T. Petsas, G. Voyatzis and E. Athanasopoulos, "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware", in *EuroSec*, Amsterdam, Netherlands, 2013.

[225] H. Gonzales, A. A. Kadir, N. Stakhanova, A. J. Alzahrani and A. A. Ghorbani, "Exploring Reverse Engineering Symptoms in Android apps", in *EuroSec*, Bordeaux, France, 2015.

[226] D. Hugenroth and F. Kilic, "Seminar Reverse Code Engineering: Obfuscation of Source Code and Intermediary Artifacts with Special Regard to the Android Platform (not published)", 2014.

[227] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu and A. Wolman, "Protecting Data on Smartphones and Tablets from Memory Attacks", in *ASPLOS 15*, Istanbul, 2015.

[228] J. Maier, "Enhanced Android Security to prevent Privilege Escalation", 16 09 2013. [Online], Available: http://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/bathesis_jmaier_final.pdf [Accessed 23 07 2015].

[229] M. Shoaib, N. Yasin and A. G. Abbassi, "Smart Card Based Protection for Dalvik Bytecode - Dynamically Loadable Component of an Android APK", 04 2016. [Online], Available: http://www.ijcte.org/vol8/1036-C040.pdf [Accessed 02 11 2015].

[230] Aktiv Soft JSC, "Guardant Code", [Online], Available: http://www.guardant.com/products/all/guardant-code/ [Accessed 07 03 2016].

[231] Aktiv Soft JSC, "Mobile Software Protection", [Online], Available: http://www.guardant.com/solutions/mobile/ [Accessed 07 03 2016].

[232] Cisco Systems, "Cisco VideoGuard Smart Card", [Online], Available: http://www.cisco.com/c/en/us/products/video/videoguard-smart-card/index.html [Accessed 15 01 2015].

[233] Several authors, "Junk byte injection in Android", StackOverflow, 13 10 2015. [Online], Available: http://stackoverflow.com/questions/33110538/junk-byte-injection-in-android [Accessed 29 01 2016].

[234] J. Jang, J. Jung, H. Ji, J. Hong, D. Kim and S. Ki Jung, "Protecting Android Applications with Steganography-based Software Watermarking", in *SAC*, Coimbra, Portugal, 2013.

[235] H. Ji and W. Kim, "Design of a Mobile Inspector for Detecting Illegal Android applications using fingerprinting", in *RACS*, Montreal, QC, Canada, 2013.

[236] S. R. Kim, J. H. Kim and H. S. Kim, "A Hybrid Design of Online Execution Class and Encryption-based Copyright Protection for Android Apps", in *RACS*, San Antonio, Texas, USA, 2012.

[237] H. K. Lee, H. S. Chung and S. R. Kim, "Memory Hacking Analysis in Mobile Devices for Hybrid Model of Copyright Protection for Android Apps", in *RACS*, Montreal, QC, Canada, 2013.

[238] Y.-S. Jeong, J.-C. Moon, D. Kim, Y.-U. Park, S.-J. Cho and M. Park, "An Anti-Piracy Mechanism based on Class Separation and Dynamic Loading for Android Applications", in *RACS*, San Antonio, Texas, USA, 2012.

[239] K.-Y. Tsai, "Android App Copy Protection Mechanism with Semi-trusted Loader", ICACT, 2015. [Online], Available: ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7224838 [Accessed 08 07 2016].

[240] Samsung, "How to protect your app from illegal copy using Samsung Application License Management (Zirconia)", 12 02 2015. [Online], Available: http://developer.samsung.com/technical-doc/view.do?v=T000000062L [Accessed 01 02 2016].

[241] J. Rädle and G. Michels, "Android Practical Course - Workshop - Hacking Android DRM (not published)", TUM, Munich, 2016.

[242] Denuvo, "Denuvo", [Online], Available: http://www.denuvo.com/#page-2 [Accessed 22 06 2016].

[243] Muhammad, "Pirates admit that games are becoming harder to crack", 07 01 2016. [Online], Available: http://www.techworm.net/2016/01/pirates-admit-games-becoming-harder-crack.html [Accessed 23 06 2016].

[244] M. Fischer, "PC-Spiel Doom: Dev-Mode ärgert Spieler, Denuvo ärgert Raubkopierer", 18 05 2016. [Online], Available: http://m.heise.de/newsticker/meldung/PC-Spiel-Doom-Dev-Mode-aergert-Spieler-Denuvo-aergert-Raubkopierer-3210042.html [Accessed 23 06 2016].

[245] M. Fischer, "Kopierschutz Denuvo offenbar umgangen: Doom, Tomb Raider und Co illegal im Netz", Heise, 08 08 2016. [Online], Available: http://www.heise.de/newsticker/meldung/Kopierschutz-Denuvo-offenbar-umgangen-Doom-Tomb-Raider-und-Co-illegal-im-Netz-3289641.html [Accessed 08 08 2016].

[246] T. Goebl, Interviewee, *Email: TU Muenchen - Denuvo Anfrage (not publicly published).* [Interview]. 01 03 2016.

[247] P. Junod, "obfuscator-llvm/obfuscator - FAQ", 17 11 2015. [Online], Available: https://github.com/obfuscator-llvm/obfuscator/wiki/FAQ [Accessed 07 03 2016].

[248] C. Mulliner, W. Robertson and E. Kirda, "VirtualSwindle: An Automated Attack Against In-App", in *ASIA CCS*, Kyoto, Japan, 2014.

[249] H. Sun, Y. Zheng, L. Bulej, A. Villazón, Z. Qi, P. Tuma and W. Binder, "A Programming Model and Framework for Comprehensive Dynamic Analysis on Android", in *MODULARITY 2015*, Fort Collins, 2015.

[250] A. Jain, H. Gonzalez and N. Stakhanova, "Enriching reverse engineering through visual exploration of Android binaries", in *PPREW-5*, Los Angeles, 2015.

[251] H. Bojino, D. Boneh and Y. Michalevsky, "Mobile Device Identification via Sensor Fingerprinting", Stanford University, [Online], Available: https://crypto.stanford.edu/gyrophone/sensor_id.pdf [Accessed 08 04 2016].

[252] S. Dey, N. Roy, W. Xu, R. R. Choudhury and S. Nelakuditi, "AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable", University of Illinois, 23 02 2014. [Online], Available: http://synrg.csl.illinois.edu/papers/AccelPrint_NDSS14.pdf [Accessed 14 04 2016].

[253] A. Das and N. Borisov, "Fingerprinting Smartphones Through Speaker", University of Illinois, [Online], Available: http://www.ieee-security.org/TC/SP2014/posters/DASAN.pdf [Accessed 14 04 2016].

[254] J. Lukas, J. Fridrich and M. Goljan, "Digital camera identification from sensor pattern noise", 12 02 2006. [Online], Available: ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1634362 [Accessed 15 04 2016].

[255] K. Kenji, K. Kenro and S. Naoki, "CCD fingerprint method-identification of a video camera from videotaped images", National Research Institute of Police Science, 1999. [Online], Available: ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=817172 [Accessed 16 04 2016].

[256] T. Filler, J. Fridrich and M. Goljan, "USING SENSOR PATTERN NOISE FOR CAMERA MODEL IDENTIFICATION", Dept. of Electrical and Computer Engineering, SUNY Binghamton, [Online], Available: http://ws.binghamton.edu/fridrich/Research/icip08paper_v7_camera_ready.pdf [Accessed 15 04 2016].

[257] M. Mohamed, B. Shrestha and N. Saxena, "SMASheD: Sniffing and Manipulating Android Sensor Data", in *CODASPY 16*, New Orleans, 2016.

[258] E. Chu, "Licensing Service For Android Applications", 27 07 2010. [Online], Available: http://android-developers.blogspot.de/2010/07/licensing-service-for-android.html [Accessed 25 06 2016].

[259] az/dpa, "IBM-Forscher entdecken neue Sicherheitslücke bei Android", Augsburger Allgemeine, 13 08 2015. [Online], Available: http://www.augsburger-allgemeine.de/digital/IBM-Forscher-entdecken-neue-Sicherheitsluecke-bei-Android-id35132027.html [Accessed 01 12 2015].

[260] D. Thomas, A. Beresford, A. Rice and D. Wagner, "Proportion of devices running vulnerable versions of Android", University of Cambridge, 2015. [Online], Available: http://androidvulnerabilities.org/graph [Accessed 01 12 2015].

[261] Lacoon, 07 2014. [Online], Available: http://www.lacoon.com/blog/2014/07/security-disclosure-googles-ios-gmail-app-enables-threat-actor/

[262] L. Tung, "Gmail app on iOS vulnerable to snooping, thanks to 'certificate pinning' flaw", 11 07 2014. [Online], Available: http://www.zdnet.com/article/gmail-app-on-ios-vulnerable-to-snooping-thanks-to-certificate-pinning-flaw/ [Accessed 25 06 2016].

[263] A. Gargenta, "Deep dive into android ipc/binder framework", in *AnDevCon: The Android Developer Conference*, 2012.

[264] revo89, "Xposed Module Repository", [Online], Available: http://repo.xposed.info/module-overview [Accessed 26 06 2016].

[265] veetip, "DisableFlagSecure", 29 09 2014. [Online], Available: http://repo.xposed.info/module/fi.veetipaananen.android.disableflagsecure [Accessed 26 06 2016].

[266] P. Schulz, "Code Protection in Android", 07 06 2012. [Online], Available: https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf [Accessed 29 04 2016].

[267] C. Linn and S. Debray, Obfuscation of executable code to improve resistance to static disassembly, 2003.

[268] P. Schulz, "Dalvik Bytecode Obfuscation on Android", DEXLabs, 21 07 2012. [Online], Available: http://www.dexlabs.org/blog/bytecode-obfuscation [Accessed 25 04 2016].

[269] A. Apvrille, "Playing Hide and Seek with Dalvik Executables", 10 2013. [Online], Available: https://www.fortiguard.com/paper/Playing-Hide-and-Seek-with-Dalvik-Executables/

[270] C. Fenton, "https://github.com/CalebFenton/simplify", [Online], [Accessed 05 04 2016].

[271] C. Fenton, "Oracle", [Online], Available: https://github.com/CalebFenton/dex-oracle/blob/master/README.md [Accessed 05 04 2016].

[272] Guardsquare, "DexGuard 7.1: Features", Guardsquare, [Online], Available: https://www.guardsquare.com/dexguard-features [Accessed 12 06 2016].

[273] obfuscer, "Can I use native compilation as Java obfuscation", StackOverflow, 20 11 2010. [Online], Available: http://stackoverflow.com/questions/4232283/can-i-use-native-compilation-as-java-obfuscation [Accessed 27 05 2016].

[274] L. Durfina, J. Kroustek, P. Matula and P. Zemek, "A Novel Approach to Online Retargetable Machine-Code Decompilation", MIR Labs, 2014. [Online], Available: http://www.mirlabs.net/jnic/secured/Volume2-Issue1/Paper24/JNIC_Paper24.pdf [Accessed 20 02 2016].

[275] Google, "ndk-build", [Online], Available: http://developer.android.com/ndk/guides/ndk-build.html [Accessed 20 02 2016].

[276] L. Broukhis, S. Cooper and L. C. Noll, "The International Obfuscated C Code Contest", IOCCC, [Online], Available: http://www.ioccc.org/years.html [Accessed 27 05 2016].

[277] R. Julien, "obfuscator-llvm/obfuscator - Installation", 30 06 2015. [Online], Available: https://github.com/obfuscator-llvm/obfuscator/wiki/Installation [Accessed 07 03 2016].

[278] C. Lattner, "The LLVM Compiler Infrastructure", University of Illinois, [Online], Available: http://llvm.org/ [Accessed 26 05 2016].

[279] Fuzion24, "Example of obfuscating an Android NDK project using O-LLVM", 28 07 2014. [Online], Available: https://github.com/Fuzion24/AndroidObfuscation-NDK [Accessed 08 03 2016].

[280] F. Gabriel, "Deobfuscation: recovering an OLLVM-protected program", QuarkLabs, 04 12 2014. [Online], Available: http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html [Accessed 08 03 2016].

[281] J. Kozyrakis, "Substrate - hooking C on Android", 01 06 2015. [Online], Available: https://koz.io/android-substrate-c-hooking/ [Accessed 28 05 2016].

[282] C. V. Bockhaven, "Intercepting Android native library calls", 2014. [Online], Available: https://cedricvb.be/post/intercepting-android-native-library-calls/#comment-9106 [Accessed 29 05 2016].

[283] S. Margaritelli, "ARM Inject", 2015. [Online], Available: https://github.com/evilsocket/arminject [Accessed 13 06 2016].

[284] K. Weiss, T. Topaloglu, N. Tslamitoas, F. Weissl and A. Abdelrahmen, "Android Practical Course SS16 - Research Task Reports Eval 2 (not published)", TUM, Munich, 2016.

[285] J. Raedl, G. Michels, V. Strelchenko and S. Al Masud, "Android Practical Course SS16 - Research Task Reports Eval 3 (not published)", TUM, Munich, 2016.

[286] H. Kirchner, J. Lucas, N. Bui, J. Hartl, A. Baus, T. Petting, T. Ladek and F. Gareis, "Android Practical Course WS15/16 - Research Task Reports (not published)", TUM, Munich, 2016.

[287] H. Grobbel, Interviewee, *Email: AW: zur Information / Zitat in Dissertation (not publicly published).* [Interview]. 07 04 2016.

[288] J. Hogenboom and W. Mostowski, "Full Memory Read Attack on a Java Card", Radboud University Nijmegen, [Online], Available: http://www.uclouvain.be/crypto/wissec2009/static/13.pdf [Accessed 14 01 2016].

[289] M. Witteman, "Smartcard Security", CHI Publishing Ltd., 2003. [Online], Available: https://www.riscure.com/archive/ISB0808MW.pdf [Accessed 14 01 2016].

[290] M. Roland, Interviewee, *Email: Re: SE (not publicly published).* [Interview]. 18 01 2016.

[291] GlobalPlatform, "GlobalPlatform Card Specification 2.1.1", 2003. [Online], Available:

http://www.win.tue.nl/pinpasjc/docs/Card%20Spec%20v2.1.1%20v0303.pdf [Accessed 16 01 2016].

[292] Sky Deutschland GmbH, "Cardsharing", [Online], Available: https://info.sky.de/inhalt/eng/unternehmen_piraterie_cardsharing.jsp [Accessed 15 01 2016].

[293] L. Francis, W. G. Sirett, K. Mayes and K. Markantonakis, "Countermeasures for Attacks on Satellite TV Cards using Open Receivers", in *Third Australasian Information Security Workshop (AISW2005)*, Newcastle, Australia, 2005.

[294] ARM, "Development of TEE and Secure Monitor Code", [Online], Available: http://www.arm.com/products/processors/technologies/trustzone/tee-smc.php [Accessed 26 06 2016].

[295] laginimaineb, "Full TrustZone exploit for MSM8974", 10 08 2015. [Online], Available: http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html [Accessed 30 11 2015].

[296] Google, "DRM", [Online], Available: https://source.android.com/devices/drm.html [Accessed 19 07 2016].

[297] Google, "Setting Up for Licensing", [Online], Available: http://developer.android.com/google/play/licensing/setting-up.html [Accessed 13 02 2016].

[298] nb, "Android Anti-Hooking Techniques in Java", 23 12 2015. [Online], Available: http://d3adend.org/blog/?p=589 [Accessed 13 06 2016].

[299] Y. Zhauniarovich, O. Gadyatskaya and B. Crispo, "DEMO: Enabling Trusted Stores for Android", CCS 2013, Berlin, Germany, 2013.

[300] Jide Co. Ltd., "Remix OS for PC", [Online], Available: http://www.jide.com/remixos [Accessed 28 07 2016].

[301] A. Killer, "Handy erkennt seinen Besitzer", BR24, 25 05 2016. [Online], Available: https://br24.de/nachrichten/Deutschland%20%26%20Welt/handy-erkennt-seinen-besitzer [Accessed 26 05 2016].

[302] N. Schmidbartl, "Analysis to identify users/devices based on different criteria and integration into a framework", 15 02 2015. [Online], Available: http://www.os.in.tum.de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/Thesis_Final._Schmidbartl.pdf [Accessed 07 04 2016].

[303] Google, "Google Playstore", Google, [Online], Available: https://play.google.com

[304] K. Olmstead and M. Atkinson, "Apps Permissions in the Google Play Store", PewResearchCenter, 10 11 2015. [Online], Available: http://www.pewinternet.org/files/2015/11/PI_2015-11-10_apps-permissions_FINAL.pdf [Accessed 07 04 2016].

[305] Google, "Best Practices for Unique Identifiers", Google, [Online], Available: http://developer.android.com/training/articles/user-data-ids.html [Accessed 16 04 2016].

[306] Google, "Location Strategies", [Online], Available: http://developer.android.com/guide/topics/location/strategies.html [Accessed 17 04 2016].

[307] Google, "Sensor", [Online], Available: http://developer.android.com/ndk/reference/group___sensor.html [Accessed 17 04 2016].

[308] P. Verest, "using SystemProperties to get Serial (Brand, Device.. etc) on Android [duplicate]", StackOverflow, 20 11 2014. [Online], Available: http://stackoverflow.com/questions/27034848/using-systemproperties-to-get-serial-brand-device-etc-on-android [Accessed 19 04 2016].

[309] unknown, "NDK android imei serial number", 21 02 2013. [Online], Available: http://www.cnblogs.com/273809717/archive/2013/02/21/2921058.html [Accessed 19 04 2016].

[310] N. Mansurov, "Dead vs Stuck vs Hot Pixels", 17 08 2011. [Online], Available: https://photographylife.com/dead-vs-stuck-vs-hot-pixels [Accessed 19 04 2016].

[311] D. Ivanov and E. Hughes, "Improving Stability with Private C/C++ Symbol Restrictions in Android N", 21 06 2016. [Online], Available: http://android-developers.blogspot.de/2016/06/improving-stability-with-private-cc.html [Accessed 28 06 2016].

[312] J. Medkeff, "photo.net", 2004. [Online], Available: http://photo.net/learn/dark_noise/ [Accessed 19 04 2016].

[313] Google, "Native Audio: OpenSL ES™ for Android", [Online], Available: http://developer.android.com/ndk/guides/audio/opensl-for-android.html [Accessed 19 04 2016].

[314] Google, "Android 6.0 APIs", [Online], Available: http://developer.android.com/about/versions/marshmallow/android-6.0.html [Accessed 08 04 2016].

[315] Google, "Ice Cream Sandwich", [Online], Available: http://developer.android.com/about/versions/android-4.0-highlights.html [Accessed 28 04 2016].

[316] Google, "IO 2011 - Evading Pirates and Stopping Vampires", 2011. [Online], Available: http://www.youtube.com/watch?feature=player_embedded&v=TnSNCX [Accessed 30 06 2013].

[317] m1m1x, "memdlopen", 26 04 2015. [Online], Available: https://github.com/m1m1x/memdlopen [Accessed 29 04 2016].

[318] J. Conrod, "Understanding Linux /proc/id/maps", 09 09 2009. [Online], Available: http://stackoverflow.com/questions/1401359/understanding-linux-proc-id-maps [Accessed 30 04 2016].

[319] E. Bendersky, "How to JIT - an introduction", 05 11 2013. [Online], Available: http://eli.thegreenplace.net/2013/11/05/how-to-jit-an-introduction [Accessed 30 04 2016].

[320] Justin Case, "[EXCLUSIVE] Report: Google's Android Market License Verification Easily Circumvented, Will Not Stop Pirates", 23 08 2010. [Online], Available: http://www.androidpolice.com/2010/08/23/exclusive-report-googles-android-market-license-verification-easily-circumvented-will-not-stop-pirates/ [Accessed 2015 05 17].

[321] Google, "Transmitting Network Data Using Volley", [Online], Available: https://developer.android.com/training/volley/index.html [Accessed 28 06 2016].

[322] T. Johns, "Securing Android LVL Applications", 01 09 2010. [Online], Available: http://android-developers.blogspot.de/2010/09/securing-android-lvl-applications.html [Accessed 26 06 2016].

[323] Bundesministerium der Justiz und fuer Verbraucherschutz, "Gesetz über Urheberrecht und verwandte Schutzrechte (Urheberrechtsgesetz)", [Online], Available: https://www.gesetze-im-internet.de/urhg/__95a.html [Accessed 28 06 2016].

[324] Apple, "Controlling Symbol Visibility", Apple, 09 10 2009. [Online], Available: https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/CppRuntimeEnv/Articles/SymbolVisibility.html [Accessed 12 06 2016].

[325] N. Douglas, "Why is the new C++ visibility support so useful?", 23 08 2013. [Online], Available: https://gcc.gnu.org/wiki/Visibility [Accessed 12 06 2016].

[326] S. Margartelli, "Using ARM Inline Assembly and Naked Functions to Fool Disassemblers", 02 05 2015. [Online], Available: https://www.evilsocket.net/2015/05/02/using-inline-assembly-and-naked-functions-to-fool-disassemblers/ [Accessed 12 06 2016].

[327] GlobalPlatform, "GlobalPlatform made simple guide: Secure Element", [Online], Available: https://www.globalplatform.org/mediaguideSE.asp [Accessed 07 06 2016].

[328] L. Stadler, "Weiterentwicklung eines Tools zur Integration von Kopierschutzmaßnahmen in Android Apps (not published)", TUM, Muenchen, 2015.

[329] Google, "Getting Started with the NDK", [Online], Available: https://developer.android.com/ndk/guides/index.html [Accessed 07 06 2016].

[330] L. Stadler, "Final presentation / Weiterentwicklung eines Tools zur Integration von Kopierschutzmaßnahmen in Android Apps (not published)", TUM, Muenchen, 2016.

[331] M. D. Ryan, "One-way secure hash functions", The University of Birmingham, 2004. [Online], Available: http://www.cs.bham.ac.uk/~mdr/teaching/modules04/security/lectures/hash.html [Accessed 07 06 2016].

[332] Ofcom, "4G and 3G mobile broadband research", 02 04 2015. [Online], Available: http://stakeholders.ofcom.org.uk/binaries/research/broadband-research/april15/Ofcom_MBB_Performance_Report_April_2015.pdf [Accessed 07 06 2016].

[333] Google, "ARA", [Online], Available: https://atap.google.com/ara/ [Accessed 27 07 2016].

[334] P. K., "A Regular expression game for Android", 30 11 2015. [Online], Available: https://github.com/phikal/regex [Accessed 13 06 2016].

[335] E. Braendle, A. Ostrovsky and T. Falkenmeyer, "Final Presentation - SIGNPOST (not published)", TUM, Muenchen, 2012.

[336] Several authors, "AndEngine", [Online], Available: http://www.andengine.org/ [Accessed 13 06 2016].

[337] E. Braendle, A. Ostrovsky and T. Falkenmeyer, "SignPost - Final Documentation (not published)", TUM, Munich, 2012.

[338] O. Pekmezci, "Implementation of a Framework for Advanced Obfuscation of an Android App by using a Secure Element (not published)", TUM, Munich, 2015.

[339] Ramps, "Android global variable", StackOverflow, 22 12 2009. [Online], Available: http://stackoverflow.com/questions/1944656/android-global-variable [Accessed 23 05 2016].

[340] T. Goebl, Interviewee, *Email: RE: TU München - Denuvo Anfrage (not publicly published).* [Interview]. 26 07 2016.

[341] M. Edel, Interviewee, *Email: AW: [RMX:ANHANG ENTFERNT] Zertifizierung/Richtlinie Kopierschutz? (not publicly published).* [Interview]. 28 07 2016.

[342] S. Bebel, Interviewee, *Email: Re: Zertifizierung/Richtlinie Kopierschutz? (not publicly published).* [Interview]. 01 08 2016.

[343] Nielsen, "MOBILE MILLENNIALS: OVER 85% OF GENERATION Y OWNS SMARTPHONES", 05 09 2014. [Online], Available: http://www.nielsen.com/us/en/insights/news/2014/mobile-millennials-over-85-percent-of-generation-y-owns-smartphones.html [Accessed 28 06 2016].

[344] C. Guitierrez, "How to view source code of dll files?", StackOverflow, 10 01 2010. [Online], Available: http://stackoverflow.com/questions/2168794/how-to-view-source-code-of-dll-files/2168826 [Accessed 22 06 2016].

[345] Sicherheitsnetzwerk-Muenchen, "Mobile Security im Unternehmen – Bericht zur Konferenz am 4.12.2014", [Online], Available: http://www.it-security-munich.net/?p=1116 [Accessed 11 05 2015].

[346] Google, "Android KitKat", [Online], Available: http://developer.android.com/about/versions/kitkat.html#44-security [Accessed 11 05 2015].

[347] S. Dent, "Google posts Windows 8.1 vulnerability before Microsoft can patch it", engadget, 02 01 2015. [Online], Available: http://www.engadget.com/2015/01/02/google-posts-unpatched-microsoft-bug/ [Accessed 23 05 2016].

[348] J. Armour, Interviewee, *Email: RE: [8-6607000004711] License Verification Library issue (or actually a general security problem due to interception possibilities) (not publicly published).* [Interview]. 30 09 2014.

[349] A. I. S. Team, Interviewee, *Email: Re: Amazon's DRM for Android (not publicly published).* [Interview]. 23 02 2016.

[350] S. S. Team, Interviewee, *Email: Galaxy App Store and its license verification library Zirconia (not publicly published).* [Interview]. 14 04 2016.

[351] Retargetable Decompiler, "Retargetable Decompiler", [Online], Available: https://retdec.com/decompilation-run/ [Accessed 16 02 2016].

[352] pyknite, cryptopathe, "Control Flow Flattening", 04 08 2014. [Online], Available: https://github.com/obfuscator-llvm/obfuscator/wiki/Control%20Flow%20Flattening [Accessed 05 04 2016].

[353] cryptopathe, pyknite, "Instructions Substitution", 04 11 2014. [Online], Available: https://github.com/obfuscator-llvm/obfuscator/wiki/Instructions%20Substitution [Accessed 05 04 2016].

[354] unknown, "Pass a string to/from Java to/from C", Real's HowTo, [Online], Available: http://www.rgagnon.com/javadetails/java-0284.html [Accessed 26 07 2016].

[355] Sven, "linux command executing by popen on C code", StackOverflow, [Online], Available: http://stackoverflow.com/questions/16127027/linux-command-executing-by-popen-on-c-code [Accessed 19 06 2016].

[356] Several authors, "Does it support Android marshmallow?", StackOverflow, [Online], Available: https://github.com/libusb/libusb/issues/188 [Accessed 06 06 2016].

[357] Infineon, "SWP SIM & UICC", [Online], Available: http://www.infineon.com/cms/en/applications/chip-card-security/swp-sim-uicc/ [Accessed 31 01 2016].

[358] N. Kralevich, Google, 2015. [Online], Available: https://usmile.at/symposium/program/2015/kralevich [Accessed 26 11 2015].

[359] unknown, "Android NDK: A guide to deploying apps with native libraries", 07 07 2015. [Online], Available: https://androidbycode.wordpress.com/tag/armeabi-v7a/ [Accessed 26 06 2016].

[360] S. Kladko, "SPA and DPA: Possible Testing Solutions and Associated Costs", BKP Security Labs, [Online], Available: http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-3/physec/papers/physecpaper08.pdf [Accessed 14 01 2016].

[361] Common Criteria, "Common Criteria", [Online], Available: http://www.commoncriteriaportal.org/cc/ [Accessed 14 01 2016].

[362] M. Rouse, "address space layout randomization (ASLR) definition", 06 2014. [Online], Available: http://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR [Accessed 19 01 2016].

[363] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003", 22 05 2013. [Online], Available: https://support.microsoft.com/en-us/kb/875352 [Accessed 19 01 2016].

[364] Xjtag, "What is JTAG and how can I make use of it?", [Online], Available: https://www.xjtag.com/about-jtag/what-is-jtag/ [Accessed 27 06 2016].

[365] Google, "StrictMode", [Online], Available: https://developer.android.com/reference/android/os/StrictMode.html [Accessed 19 06 2016].