DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Johannes Neutze, B. Sc.

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis of Android Cracking Tools and Investigations in Countermeasures for Developers

Analyse von Android Crackingtools und Untersuchung geeigneter Gegenmaßnahmen für Entwickler

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

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

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

Acknowledgments

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

Assumptions

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

Abstract

`http://users.ece.cmu.edu/~koopman/essays/abstract.html` Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Contents

Glossary

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

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

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

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

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

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

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

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

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

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

Acronyms

.dex Dalvik EXecutable file.

.jar Java Archive.

.odex Optimized Dalvik EXecutable file.

ADB Android Debug Bridge.

ADT Android Developer Tools.

AOT Ahead-Of-Time.

APK Application Programming Interface.

APK Android Application Package.

ART Android RunTime.

DRM Digital Rights Management.

DVM Dalvik Virtual Machine.

ELF Extensible Linking Format.

GC Garbage Collection.

IP Intellectual Property.

JIT Just-In-Time.

JNI Java Native Interface.

JVM Java Virtual Machine.

LLVM Low Level Virtual Machine.

LVL	License Verification Library.
NDK	Native Development Kit.
OS	Operating System.
OTG	USB On-The-Go.
SDK	Software Development Kit.
SE	Secure Element.
TEE	Trusted Execution Environment.
VM	Virtual Machine.

1 Introduction

1.1 Licensing

- legally binding agreement between two parties
- rights of the licensor and the licensee
- protect the software creator, commercialize it as a product
- boundaries of usage for the user and prevents him from illicit usage
- -
- different variants
- full feature costs, often subject of piracy
- mechanisms are implemented to enforce the legal agreement
- Digital Right Management
- -
- pirate try to circumvent
- race of arms

1.2 Motivation

- market share, #devices
- google play, apps, revenue
- becoming more important
- better apps attract pirates
- developers aware, many have problems
- The scope of this thesis is to analyse Android cracking applications, like Lucky Patcher, and to investigate in countermeasures for developers.

1.3 Related Work

- andere arbeiten
- muntean, protecting against luckypatcher
- kovacheva, efficient obfuscation
- bernhard, protecting license verification library

2 Foundation

- explain what this chapter is about

2.1 Software Piracy

- schaden von piracy
- wodurch schaden entsteht

2.1.1 Developers

- direct revenue loss by piracy
- follow up losses
- cracked inapp purchases (freemium, pro version)
- inapp ads
- cannot support app anymore when in blackmarket
- no updates, security, etc
- bad app behavior falls back to developer
- no money no reason to continue developing

2.1.2 Users

- also bad for users, harmed by piracy
- download it because free
- can be altered, background process, e.g. malware, permissions
- bad stability, missing updates

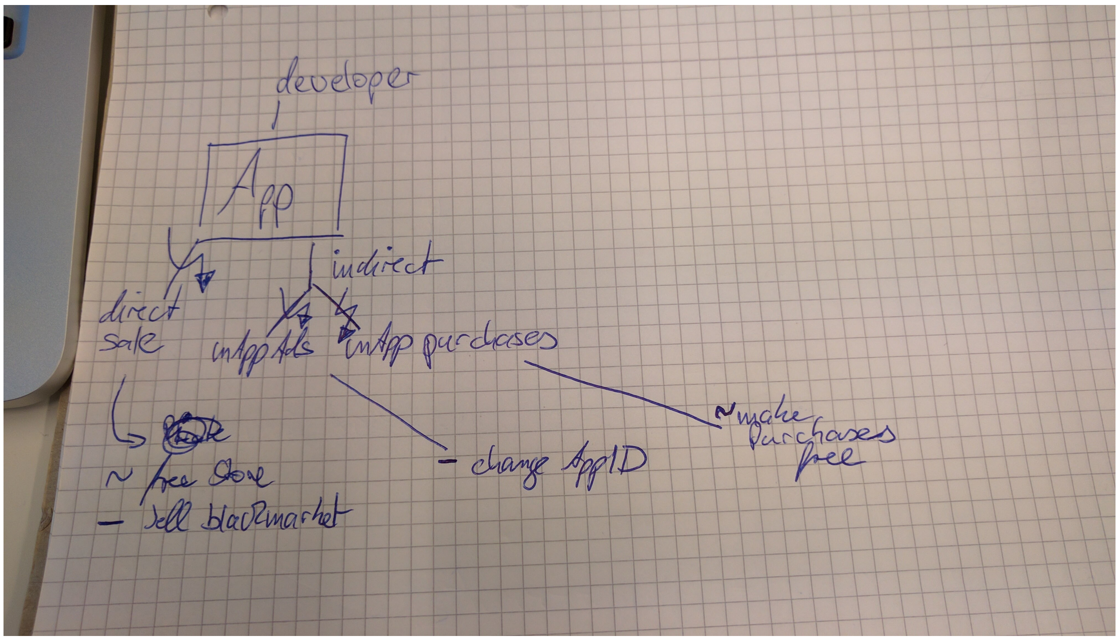


Figure 2.1: Different ways to generate revenue

- worse user experience
- do not install without deep inspection

2.1.3 Piracy on Android

- piracy wide spread, especially china
- easily found on the internet, blackmarket, crack sites
- declared as "free premium apps", older versions
- only possible if license mechanism cracked
- example today calendar, piracy rate
- some developers gabe up

2.2 Android

- what is android?

- mobile os, 2007, google
- linux kernel
- mobile devices, wearables
- limited system resources
- arm
- overview architecture
- kernel lvl
- dalvik/art and native libraries
- application framework
- application lvl
- abstract, so android can run on different ahrdware

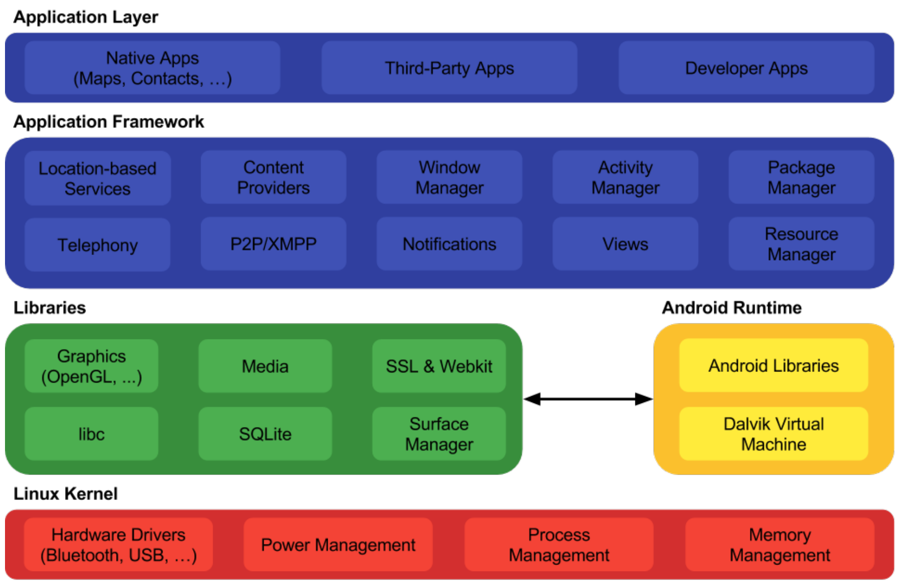


Figure 2.2: Android's architecture [21]

2.2.1 Android Application Package (APK)

- uses apks from stores, downloaded oder developed
- zip format
- build process
- written in java, similar to java build process, java compiler, obfuscator, packed to jar file
- java byte code to dalvik byte code, described later, obfuscation
- three main parts, classes.dex, resources, manifest, combined to archive file
- signing
- structure apk file

2.2.2 Dalvik Executionable File Format

- android uses dex, executed by dvm
- compiled from java bytecode
- different, stack vs register for arm
- direct mapping from code to registers, uses 32 bit register, 64 bit use adjacent registers
- bitcode less compact, 16 bit instead of 8bit in java
- 218 opcodes, dest source ordering
- in java heterogenous pools, dx for dalvik merges, reduce dublicates, best for strings
- all classes into one file as well
- optimization, dex odex same semantics
- downside, easy to reverse engineer

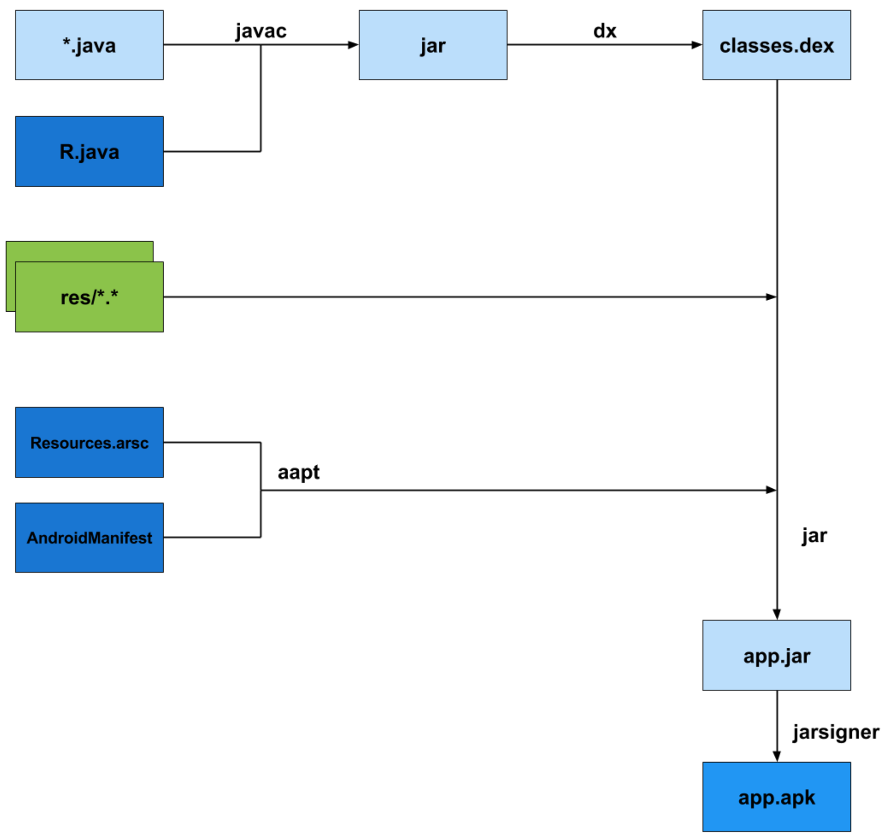


Figure 2.3: Android Application Package (APK) build process [19]

2.2.3 Installing an APK

- install before running
- two steps, verification and optimization respectively art
- differences dvm, art next section, useful because different processors
- classes.dex to odex and put to dalvik cache, done once then odex always used, improves startup time, DVM
- more complex compilation, ART
- when run later, sandbox, each applicaiton own id

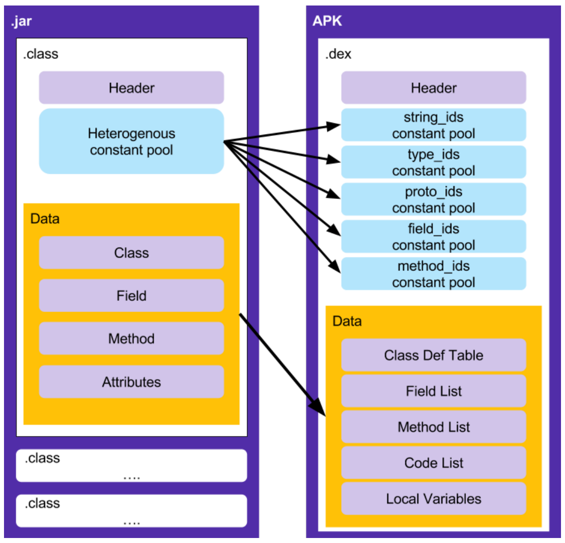


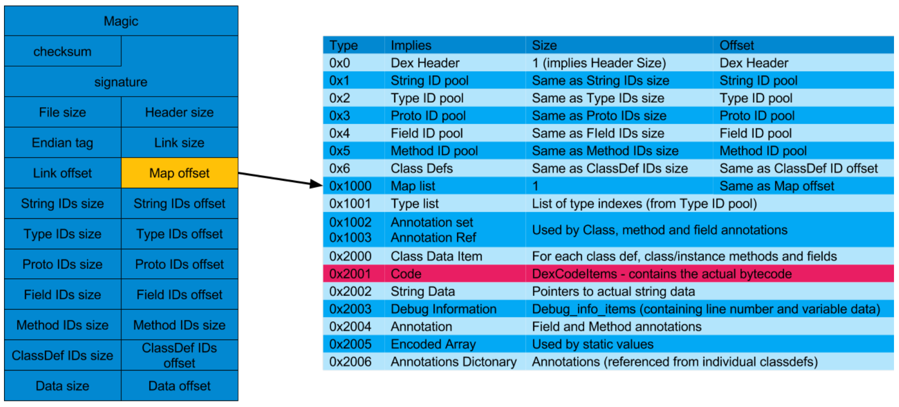
Figure 2.4: Java Archive (.jar) to APK transformation [9]

2.2.4 Dalvik Virtual Machine

- allgemeines dvm, wer wann name
- constraints are reason for way it works
- customized, optimized apachy harmony
- based on Java, but not fully compatible
- 16bit opcodes & register vs 8bit and stack
- advantages of register, low power but larger code and fetching times
- optimized memory sharing, optimized startup times
- last change 2.2 jit

2.2.5 Android Runtime

- introduced 4.4, optional, for developers
- works with dex standard
- 5.0 runtime of choice, dvm major flaws



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

Figure 2.5: Dalvik EXecutable (.dex) file format [19]

- until 6.0 evolving, breaking older versions, few documentation
- designed to address shortcomings, jit worse than native code, each time jit is wasteful, background threads require more cpu, slow more battery, jitter, 32 bit only
- 64 bit better, improvements in vmm hit to aot, less overhead cycles, non blocking, fore+background threads
- art can compile to native or llvm, each has purpose and advantage, speed vs compability, native prefered
- two file formats
- .art similar to zygote, proprietary, poorly documented, still changing, maps in memory before linked oat, contains pre-initialized classes and objects
- .oat, one master file boot.oat contains "best of" android framework jar, still ahve .odex but are elf/oat, inside dalvik cache, dex of apk embedded
- art supports multiple architectures
- lessons: base is dex so still 32 bit, no 64 bit registers and only few 64bit instructions, code is not always efficient, room for improvement, codeflow is same, not all methods guaranteed to be compiled, reversing a pain

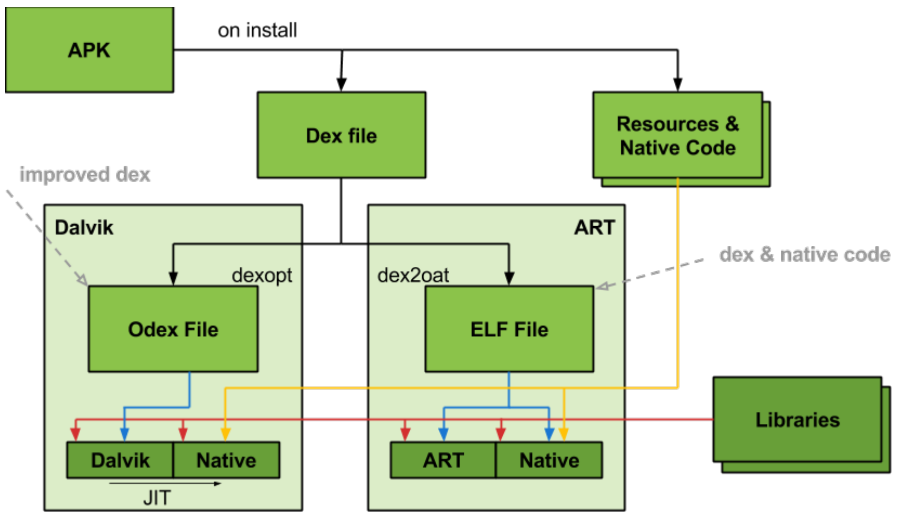


Figure 2.6: Installing an APK on a device [7]

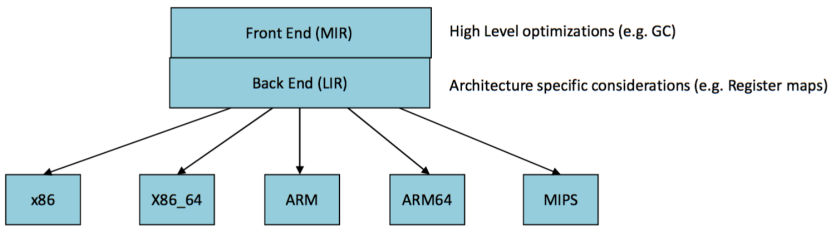

Front End (MIR)
High Level optimizations (e.g. GC)
Back End (LIR)
Architecture specific considerations (e.g. Register maps)
x86
X86_64
ARM
ARM64
MIPS

Figure 2.7: artarch

2.2.6 Root and Copy Protection

- root is getting total control over system
- comes from linux
- used to circumvent limitations by carrier, manufacturers, extend system
- not approved but vulnerabilities used to exploit, often and many exploited
- easy even for non techies, videos tutorials tools
- su user helps to handle
- risks, bricking

- root voids copy protection since access to asec folder can be gained, effective when root was hard to get

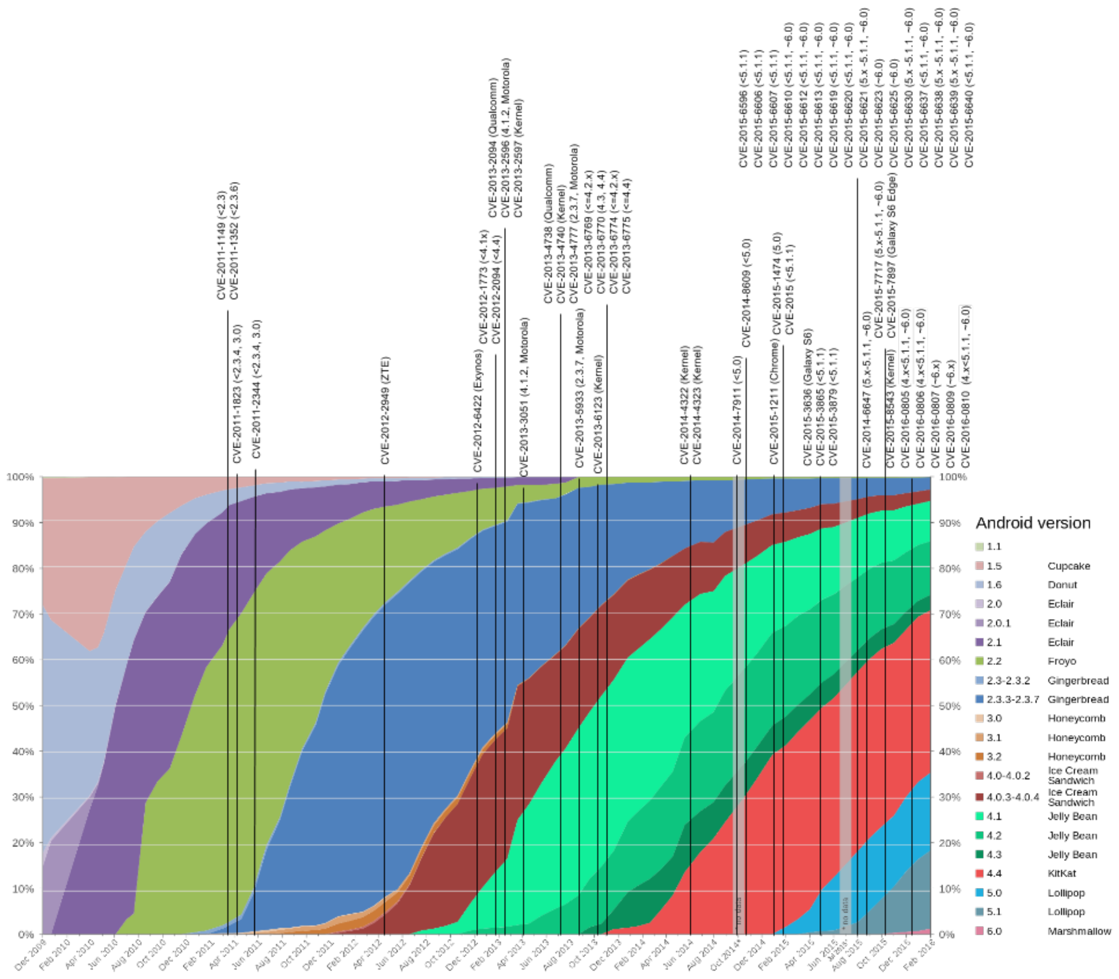


Figure 2.8: Timeline of market share of evolving Android versions and dates of root exploits recorded [13] [1] [11] [12]

2.3 License Verification Libraries

- since copy protection does not work anymore, new methods to prevent piracy
- not only relevant to google

- since unknown sources and other stores, others want money from android as well
- amazon and samsung introduce own stores, have same problems
- others are not so big that relevant, china as biggest player does not use protection
- need to give developers a safe environment
- each implements a drm

2.3.1 Google's License Verification Library (LVL)

- by google to fix copy protection
- 27.7.2010
- easy to use, documentation

Functional Principle

- flow überprüfen...
- query trusted server
- manually implemented by developer, code is provided
- simple checking and callback with google
- library manages all, sends request to server, drm, does not interfer with app logic
- developer handles result while library handles rest, full control of what happens with callback
- additional information send, licensechecker gets packagename, nonce, callback
- play service adds google account
- server checks whether purchased and responds
- security is important, public/private key encryption provided by google, tampering protection
- requires online connection, google play services at first time
- when how often done by developer
- replaces old copy protection

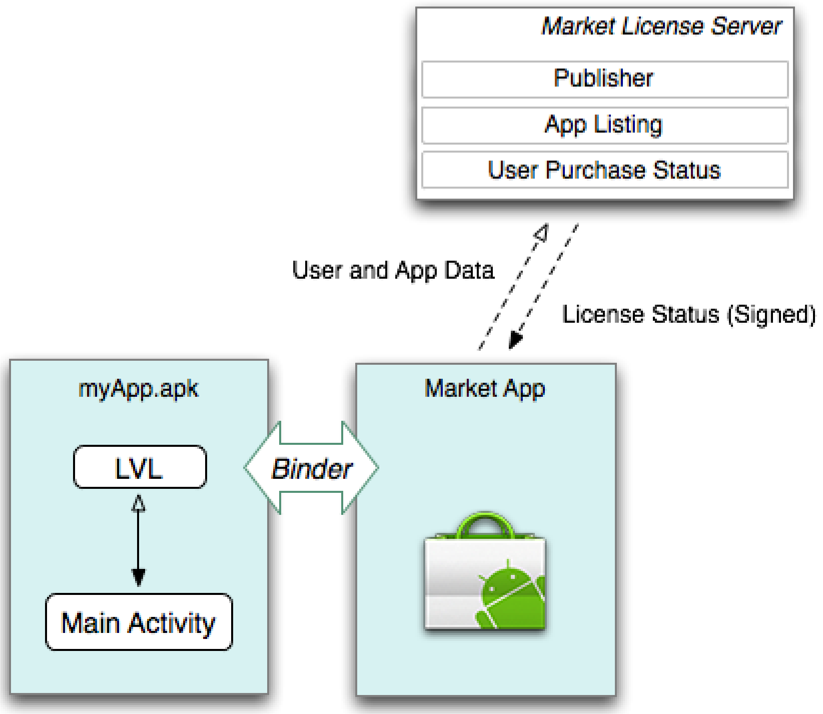


Figure 2.9: Google's implementation of license checking [6]

Implementation of the License Verification Library

- implemented in few steps
- publisher acc needed to use lvl, create app to get key
- extract lvl from sdk, set permission
- create elements
- implement callback
- freedom where to implement everything

2.3.2 Amazon DRM (Kiwi)

- introduced with amazon app store, 22.3.2011
- own drm since lvl only works with google play

- called kiwi
- developer is free to use

Functional Principle

- account on amazon
- instead of implementing, wrapped when uploading
- decompiled server sided and injected by amazon, resigned
- requires user to be loged into amazon store, communication with amazon app directly, not over internet, no permission required

Implementation of Kiwi

- wrapper injected by amazon
- same logic in every app
- reengineering shows that it replaces the onCreate, called first then the original one
- library code is obfuscated
- logic does not interweave with app logic since injection is wrapped around, else would need developer

2.3.3 Samsung DRM (Zirconia)

- samsungs solution for their store
- called zirconia

Functional Principle

- same drm as other libraries
- works only on samsung since download requires samsung store
- checks license with server, unique for each device/application so copying does not work

- process: query for stored license, if yes: execute, if not exist or invalid send info about device and application to server (first time only), server responds, app stores, next time from begin
- asynch callback
- does not work without network or airplane if license not found, does not rely on galaxyapps like google, direct communicaiton
- do nto call to often since internet call is slow

Implementation of Zirconia

- jar and native libraries have to be added
- permission for internet and phone state required
- easy implementation, anywhere in application

2.3.4 Overview License Verification

- looks like solution
- does not prevent from copying, but implements mechanism independent
- all work by talking to other instance, evaluating result
- based on yes/no answer
- google started, others followed since they want developers to come to store
- since dex, can be decompiled and analyses then attacked, even cracking solutions available for the big players
- developer has to decide between reach and security, unknown stores are not likely to be attacked but developer cannot sell apps that well, but if the app attracts attention, even these unknown stores will be cracked

2.3.5 Abstraction

2.4 Code Analysis

- cracking tools alter apps by applying patch on apk, onylthing they can touch, so analyze apk

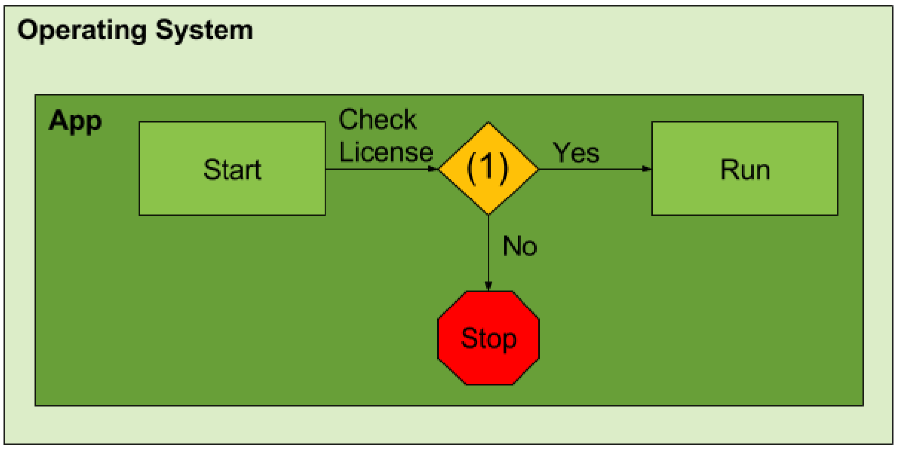


Figure 2.10: Abstraction of the current license verification mechanism. The library is represented by (1)

- static tools since code is modified, get overview how to find countermeasures
- different layers of abstraction, bytecode (what exactly is changed), smali (what function is changed), java (functionality changed), java problems since decompiler
- code dependecy, dex java isomorphic, differences
- overall decompilation
- best case get detailed overview of patching mechanism on high level, since readable, maybe low level sufficient, understand relation between change and function
- different tools and techniques

2.4.1 Retrieving an APK

- installed /data/app
- can be pulled when package is know, can be aquired with packagemanager
- download to computer with adb pull
- with root file explorer can access directly
- licensetest is appname placeholder

2.4.2 dex Analysis

- highest abstraction, bytecode
- luckypatcher has only this as well, applies patches to it, raw presentation
- inside classes.dex, contains applicaiton bytecode
- hexdump to the classes.dex to receive readable bytecode
- done by using this scrypt
- result looks like this

2.4.3 Smali Analysis

- frage: minimal halten oder auch was groß zu smali erzählen?
- smali is most popular decompiler for dex, used by many tools
- readable version of dex, similar to jasmin syntax, bijective
- makes changes more understandable
- this script was used
- output looks like this

2.4.4 Java Analysis

- java code is best level to understand changes since it is programmed on this level
- reversed build process, on compilation some information is always lost, cannot be recovered when decompiling
- dex uses different patterns since for mobile usage, java cannot interpret them so good, this is the reason to use two different decompilers
- androguard, script, outcome
- jadx, script outcome
- difference between those two

Androguard

- was macht androguard
- wie sieht output aus

JADX

- was macht androguard
- wie sieht output aus

—-

- compare

2.4.5 Detect Code Manipulations

- diff is used to compare reengineered source code of original app with cracked app, short diff description
- find differences and identify pattern, understand how luckypatcher is working, compare on each abstraction level
- explain script
- explain result

3 Cracking Android Applications with Lucky Patcher

- shortly explain cracking apps and what they are used for
- luckypatcher most popular that is why analysed

3.1 Lucky Patcher

- what is luckypatcher, what is written about it on the site, who created it
- lucky patcher is used to crack, remove ads and permissions
- works with and without root
- indirect patching without root
- root and busybox when direct patching
- requires no technical knowledge since automatic, popular
- focus on bypassing license verification, works as legally aquired
- does not attack server connection but application code
- how does applying a patch work, does not guarantee success
- analysed in two ways, code and cracked applications

3.2 Code Analysis

- 6.0.4 is analyzed using androguard and jadx, reversed code inspected in editor
- structure analysis, 4 categories, support library, luckypatcher code, libraries, patched lvl - additional assets
- difficult analysis since no real strucutre

- launcher activity can be found, calls fragment, 17k lines of code, hard to analyse since obfuscated
- others already tried to analyse code

3.3 Analysis of Patched Applications

- code analysis nto successful, analyse modified applications
- find attack points by using different apps and patching modes
- analysis with tools explained before
- use apks not odex since odex may be optimized
- goal is to see changes on different levels, dex, smali, java with diff
- start with reference application to ahve full control, lvl, samsung, amazon
- also on different apps to have a variety
- A pattern is a set of predefined sequences of bytecode in which a certain values are modified.
- execute all modes to identify all patterns
- explain modes as described in LP

3.4 Patching Patterns

- common things like where the name comes from, how the proceeding is, naming convetions
- explain class attacked, what is it good for
- explain dex change
- explain what this dex change is in smali
- explain what the java change is
- explain the functionality change
- overview which mode uses which pattern

3.5 Conclusion and Learnings

- result of patching, which applications work in which modes on unrooted devices
- since this can be tested it means they can also be sold on blackmarkets
- explenation of results, lvl (most time), amazon (always), samsung
- comprehension of luckypatcher
- instead of patching inital call, library itself is patched since it could call necessary code
- known bytecode patterns, replace with custom, prevent automatic patching
- in addition: does only attack library -> tampering checks?
- amazon, samsung cannot be fixed inside library, company has to act
- abstact presentation how attack works
- easy to attack since always this pattern if(someRights != null) // you have granted rights. else // You don't have any rights for the feature in cause. Try // some features. (Currently not supporting multiple 'features') which can be attacked easily by voiding one result (see bytecode)
- the result is unary check
- since many applications are succesfully attacked with automatic mode, high motivation to modify library to be unique

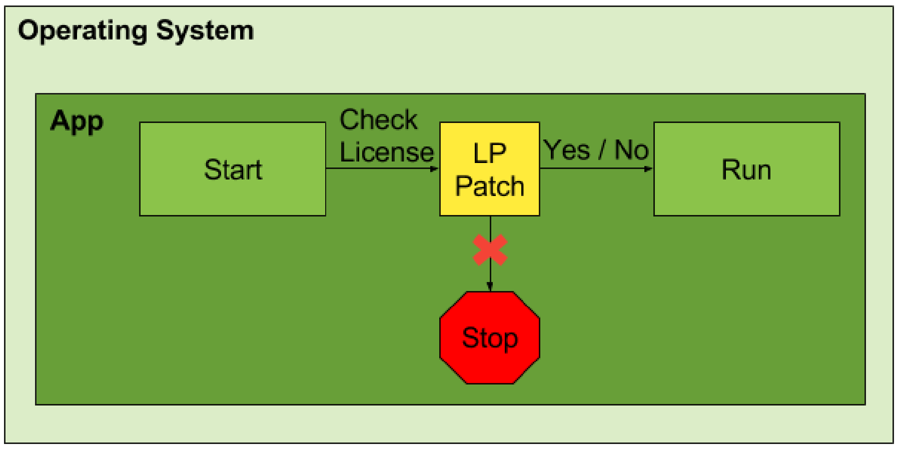


Figure 3.1: Abstraction of the current attack on the license verification mechanism

4 Countermeasures for Developers

- after analyzing aptterns, try to use knowledge to fortify
- explain what happens in chapters

4.1 Extend Current Library

-

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

4.1.1 Modifications on the Google LVL

- success of LP because no one customizes
- use it against casual piracy
- do not know where to fortify
- two proposals: actively in library, native

Modify the Library

- LP relies on patterns applied to interfer logic
- solution: prevent to apply patterns, opnly possible when code can be modified (lvl), more difficult for amaozn and samsung
- modification also increases complexity and thus harder to reverse
- three locations:
- core licensing library logic

- entry andexit points of the licensing library
- invocation and handling of the response
- Ideas for each

4.1.2 Native Implementation of LVL

- LP automatic mode attacks dex code
- android supports native code, implement lvl in native code
- usually for cpu heavy tasks, increased complexity, desired when lvl, harder to understand and to decompile
- two scenarios, implemented by developer and by google
- developer: unique but requires knowledge, skill and time
- attackers have to invest more time to analyse since native code and unique, costs a lot of time, scares off attackers, but iof cracked it can be provided as custom patch
- google: if native, harder to find vulnerabilities since code more complex
- take some time to crack but motivation since all apps would use it, provide as custom patch
- to counter this: heavy obfuscation, encryption, dynamic code generation, use all what possible to make as ahrd as possible to crack
- good idea, but hard to implement as noone has done it yet, but first steps to address vulnerabilities of dex have been done with art
- in general: can also be cracked and provided as custom patches, but more time has to be spend since analysis is more difficult, but luckypatcher already contains custom patches so people do it

4.1.3 Tampering Protection

- check environment for integrity
- integrity broken when app is cracked, unless very precisely done
- can be detected and implemented as additional checks, prevent app from running

- can be circumvented with patching as well, but attacker has to know that they exist and where they implemented in order to patch them
- obfuscation and spreading helps, or native implementation
- does not rpevent LP from working but adds another layer of security

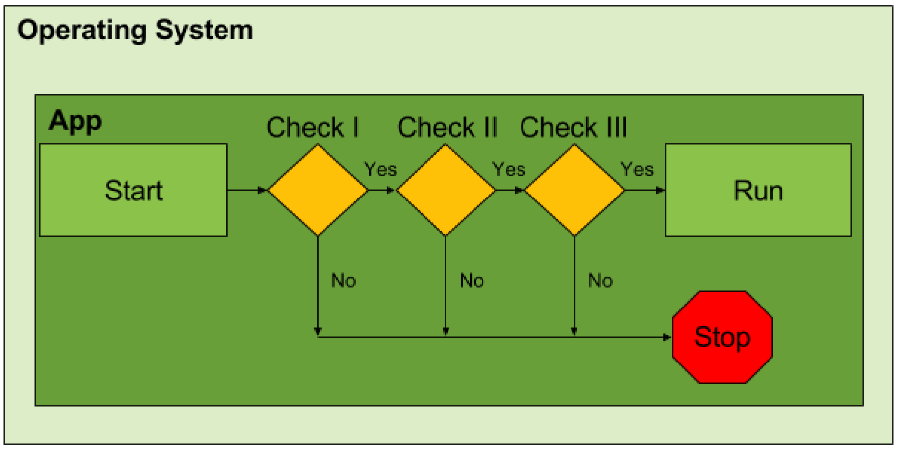


Figure 4.1: Introduction of additional tests to check environment and integrity of the application

Debuggability

- debug is needed to gain additional information when analysing the app
- prevent app from running when debug is active

Root

- root can be used to crack applications on phone
- prevent app from running when root is detected
- be careful, some people like to have root for other reasons than piracy
- google has similar framework, safetynet

Lucky Patcher

- luckypatcheer is used to attack applications
- do not run when LP detected
- can be extended to other software as well
- already implemented in library antipiracysupport

Sideload

- cracked apps can be on phones even though lp or root are not present
- installed from e.g. blackmarket or traded
- normaly apps with lvl can only be installed from store since they are only distributed there
- prevent to run the app when installed from other source
- google notes that this feature is only possible by using an undocumented method, handle with care

Signature

- when the code is changed, the checksum has to be adjusted and thus the signature has to be redone as well
- lp does not have the developers signature
- detect if signature is changed (app is very likely to be altered then) and do not allow to run
- similar thing does google with their maps license

Flow Control

- since the code is known and can be modified, this can be used to implement functionality inside that Library by reusing methods or implementing calls which are only called if the right flow inside the library is done, e.g. put important method in block that is skipped when patched
- implement second version of LVL that is known to fail, LP will patch both, stop app when failing licensecheck is true

4.1.4 Obfuscation

- modification prevents automatic patching, manual analysis and custom patches still possible
- use obfuscation to make reengineering harder and time consuming, does not offer total protection
- helps since attacker has to understand what happens in order to fix it, obfuscation removes names so understanding is harder
- applied when compiling
- definition obfuscation
- can be applied everywhere without much extra workload
- does not prevent automatic patching or alters flow
- removes symbols that reveal information
- can be applied to soruce code or bytecode, bytecode needs advanced algorithms
- number of commercial and open-source obfuscators from java
- different dex obfuscators
- some emthods cannot be obfuscated since they need to communicate with the android frmework
- does not protect directly against LP but helps versus attacks on customized libraries

Proguard

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

A Java source code obfuscator. ProGuard performs variable identifiers name scrambling for packages, classes, methods and fields. It shrinks the code size by automatically removing unused classes, detects and highlights dead code, but leaves the developer to remove it manually [18]

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

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

after proguard is finished dx converts to classes.dex

[20]

identifier mangling, ProGuard uses a similar approach. It uses minimal lexical-sorted strings like a, b, c, ..., aa, ab, original identifiers give information about interesting parts of a program, Reverse engineering methods can use these information to reduce the amount of program code that has to be manually analyzed -see- neutralizing these information in order to prevent this reduction, remove any meta information about the behavior, meaningless string representation holdin respect to consistence means identifiers for the same object must be replaced by the same string, advantage of minimizing the memory usage, e development process in step "a" or step "b"
string obfuscationa, string must be available at runtime because a user cannot understand an obfuscated or encrypted message dialog, information is context, other is information itself, e.g. key, url, injective function and deobfuscation stub which constructs original at runtime so no behaviour is changed, does not make understanding harder since only stub is added but reduces usable meta information
[23]

ProGuard is an open source tool which is also integrated in the Android SDK, free ProGuard is basically a Java obfuscator but can also be used for Android applications because they are usually written in Java // feature set includes identifier obfuscation for packages, classes, methods, and fields was kann er noch? -see- Besides these protection mechanisms it can also identify and highlight dead code and removed in a second, manual step Unused classes removed automatically by ProGuard. easy integration[4]

optimizes, shrinks, (barely) obfuscates, , reduces size, faster removes unnecessary/unused code merges identical code blocks performs optimiztations removes debug information renames objects restructures code removes linenumbers, stacktrace annoying [14] [27]

Dexguard

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

commercial Android obfuscator for bytecode and source code various techniques including strings encryprion, encrypting app resources, tamper detection, removing logging code [18]

son of proguard, does everything that proguard does its a optimizer and shrinekr, obfuscator/encrypter, does not stop reverse engineering automatic reflection, string encryption, asset/library encryption, class encryption(packing), applciation tamper protection, removes debug information may increase dex size, memory size; decrease speed [27]

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

protects apps from reverse engineering and ahckign attacks makes apps smaller and faster specialized fr android protects code: obfuscation, hides sensitive strings,keys and entire algorithms [16]

Allatori

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

commercial Android obfuscator name obfuscation, control flow flattening/obfuscation, debug info obfuscation, string encryption the result is a decreases dex size, memory, increases speed, removed debug code+ like Proguard+string encryption [27]

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

[2] [3]

4.2 Complex Verification

- big vulnerability is binary outcome,predictable, can be forced unary and
- wrong result will be accepted
- improve this flaw by introducing complex mechanism and replace old
- two proposals: content server with verification on server to counter modification, encryption to make outcome unpredictable

In order to fight this major flaw, more complex mechanisms have to replace the original license verification. In the following, different approaches are proposed. The first proposal is the introduction of content server which allows to move part of the functionality, like the license verification, from the application to a server. The server cannot be attacked by the cracking application and thus is safe.

The second proposal is to introduce encryption. Encryption enables unpredictable outcomes instead of binary ones.

4.2.1 Content Server

- register on server, code cannot be altered, content only delivered when user verified
- hacking around is not successful since the applciation does not work without the server

4.2.2 Encryption

Another approach to is the introduction of more complex license verification mechanism by using encryption. The advantage of encryption is the unpredictable outcome of an input which is not spoofable anymore compared to the binary check. Before taking advantage of encryption, it has to be decided what content should be encrypted and where the key should be stored.

Encryption

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

Resource Decryption

The first approach is to apply encryption on the application's static resources. This includes the application's hard coded strings as well as image assets. Whenever a resource is used, it has to be decrypted first. The increase in security comes at the cost of decreased performance. As long as application critical strings, like server addresses are encrypted, the application is unable to work. In case no critical strings are present, the application will work as usual, but the user experience will be not sufficient because all strings are still encrypted and thus have no meaning. Figure 4.2 shows the abstract implementation of resource decryption.

Action Obfuscator

The second approach is to use encryption as obfuscation. The idea is to have a method, which is called by each method, to deligate all calls according to an encrypted parameter. The attacker can try to guess which method call is linked to each method, but when

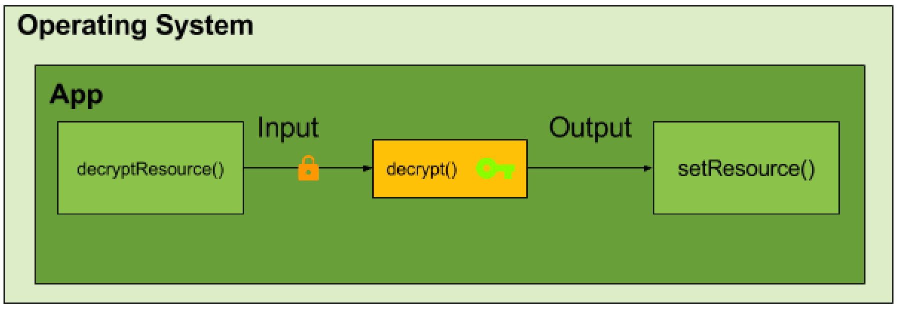


Figure 4.2: Encrypted resources which have to be decrypted on startup

additional encryption for the parameters is applied and decryption is used in the target method, it is very hard to crack the logic without modifying a lot of code. In order to make it harder to circumvent the encryption of the methods, objects can be encrypted as well. An abstract presentation of the mechanism can be seen in figure 4.3.

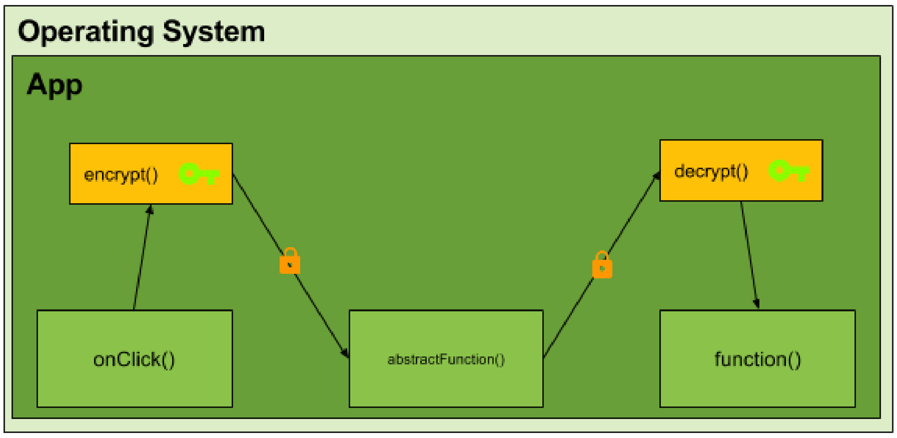


Figure 4.3: Encrypted actions to obfuscate dependencies

Communication Decryption

The third approach is to use encryption on the server response as seen in figure 4.4. This is an additional security feature which is applied in combination with a content server described in subsection 4.2.1. When the user does the login on the server, additional unique device specific paramters have be passed as well. On the first login, the server generates a cryptographic key which is used for communication with the user on this specific device. The corresponding key can either be generated on the device or be shared by the server. This mechanism allows only authorized users on a specific device to decrypt the communication.

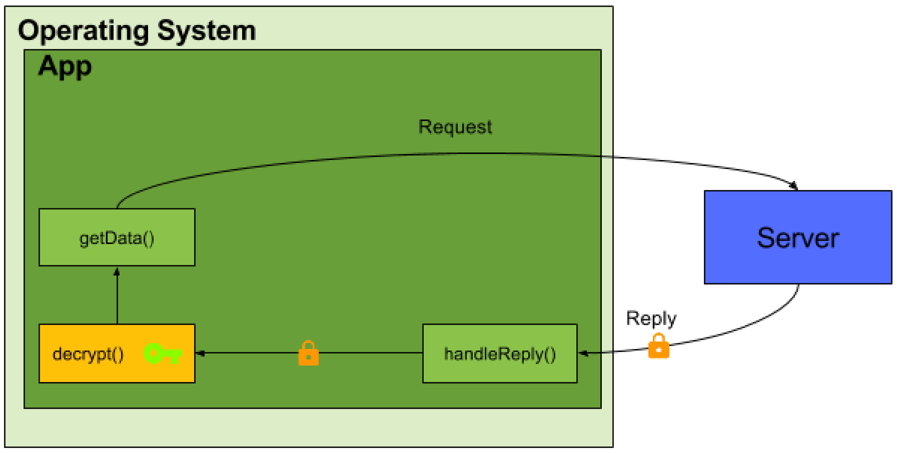


Figure 4.4: Encrypted communication with a server

A similar approach is used for streaming Digital Rights Management (DRM) protected content on Android. The encrypted content can only be decrypted by a native interface provided by the Operating System (OS) which stores the decryption key. [5] This methodology focuses on the security of the content instead of the application itself.

Encryption

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

Secure Element

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

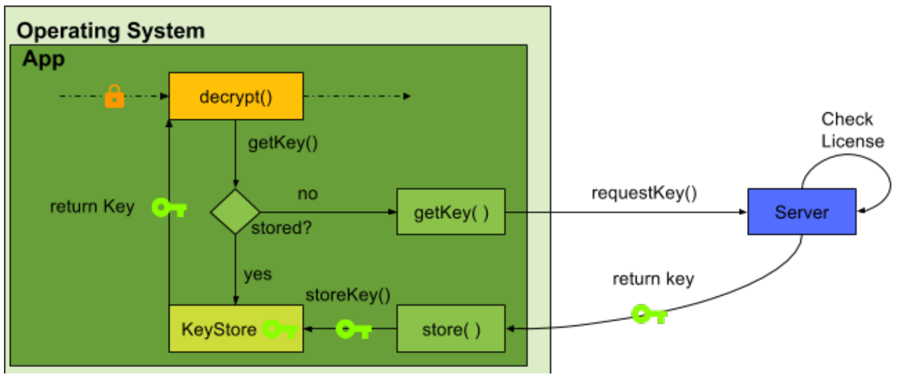


Figure 4.5: Retrieving the key after successful identification from the server and store it local on device

Secure Element

Since there is the possibility to read the cached encryption key [26] and crack the encryption, the use of Secure Element (SE) is proposed. A secure element is a tamper-resistant platform which can be used to securely host applications and cryptographic keys [15]. There are different form factors for SEs. For Android, the microSD form factor is the interesting one. It can be either mounted in the microSD card slot or by using an adapter on the USB interface, which requires the device to support USB On-The-Go (OTG) [**usbOtg**]. The resource is accessed over reads and writes to the filesystem. Since the SE has to be small to fit the size of an microSD card and powered by the host system, its hardware capabilities are constrained. The result is a performance of 25MHz which does not allow complexe comnputations. [25]
For this reason the usage of the SE is restricted to simple tasks, like storing a key used for decryption. The advantage of an SE is that its functionality is outside of the Android application and thus cannot be manipulated by Lucky Patcher.
An abstract presentation of the use of a SE can be seen in figure 4.6.
Integrating a SE comes with some problems.

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

The first problem is that the user is required to buy extra hardware. This means the user has to spend extra cache and to have the SE always around. The connection to the device using a cable is not the most convenient solution as well. The second

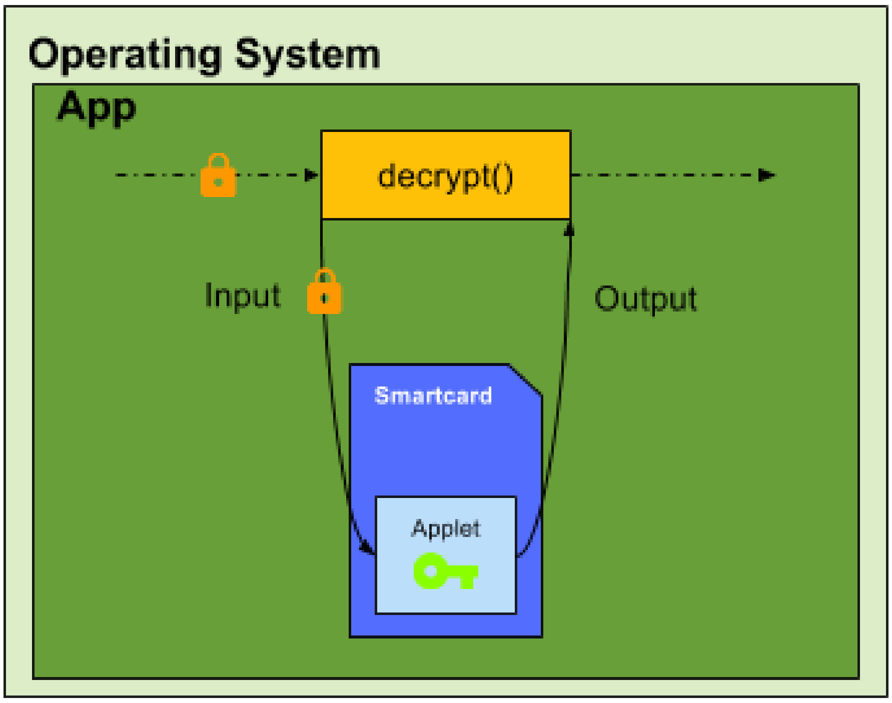


Figure 4.6: Decryption by using a smartcard

problem is that not all devices either have an microSD card slot, nor OTG support. For example, the Nexus 7 (2012) and the Nexus 6P neither have the capability to use a microSD card. While the Nexus7 was supposed to have OTG, it did not work with the used SE, while the Nexus 6P did not support OTG out of the box. Both devices needed even needed additional plugins to read the OTG mounted microSD in a file explorer. The third problem is that each manufacturer implements its own interpretation for the interface which makes SE incompatible to each other. For this reason, the SD Association proposed the smartSD in order to have a universal standard for SEs [24].

solution which are out there have major security flaws, smartcard itself can be attacked

have problems on their own, nur so sicher wie das secure element DAP Verification .... normalerweise muss jede Applet, die auf so ein Secure Element/Smartcard etc. kommt mit ner Signatur unterschrieben sein ...

Waehrend ich Exploits finden konnte, die Dir erw. Zugriff geben, wenn du Applets installieren kannst, u.a.

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

•

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

4.3 Additional Environmental Features

•

since dex code as we have seen is vulnerable to attacks and reeengineering, the solution could be outside of the application, provided by either the environment or another hardware

level tiefer

4.3.1 Trusted Execution Environment

•

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

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

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

example Trustzone, Knox

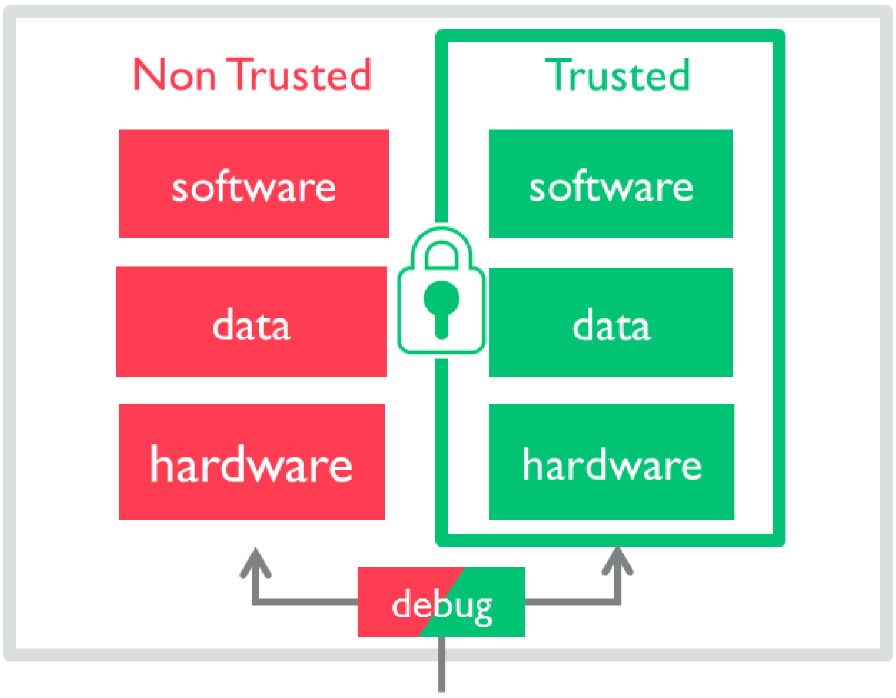


Figure 4.7: tee [8]

[10][8]

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

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

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

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

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

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

[10][8]

not accessible to anybody in general many different solutions, focus should be on one unique standard in order to fix the problems and make compatibility , google already started by integrating features of samsung's knox into android lollipop [22]

4.3.2 ART

-

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

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

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

already on the way, cannot be done from one day on the other, but right now not a protection against luckypatcher, will only be a solution when art code included in apks but why not now? Evaluation Why is Android not all ART now? Your applications still compile into Dalvik (DEX) code, Final compilation to ART occurs on the device, during install, Even ART binaries have Dalvik embedded in them, Some methods may be left as DEX, to be interpreted, Dalvik is much easier to debug than ART
[19]

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

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

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

When creating odex on art it is directly put into art file
[19]

5 Conclusion

- test

research and also a valuable market for companies
Because source code can be easier recovered from an application in comparison to x86, there is a strong need for code protection and adoption of existing reverse engineering methods. Main parts of Android application functionalities are realized in Dalvik bytecode. So Dalvik bytecode is of main interest for this topic
Also, the Android system does not prevent modification of this bytecode during runtime, This ability of modifying the code can be used to construct powerful code protection schemata and so make it hard to analyze a given application.
[23]

current state of license verification on Android reverse engineering far too easy due to OS, extract/install allowed gaining root easy, allows everyone especially pirates avoiding proteciton mechanisms java was chosen to support a lot of hardware, java has bad protection

lvl popular but broken, has not done much since beginning of known issues [20]

auch wichtig weil wenn crackable dann upload zu stores und dann malware http://www.hotforsecurity.com/blog/mobile-app-development-company-fights-off-android-malware-with-obfuscation-tool-3717.html

5.1 Summary

- test

jedes chapter beschreiben

5.2 Discussion

- test

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

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

eternal arms race no winning solution against all cases, jsut small pieces quantitative improvement no qualitatively improve resilience limited to quantitative resilience, matter of time until small steps generate more work for reengineering, ggf lower motivation for cracker only matter of time until patching tools catch up, completely new protection schemes need to be devised to counter those [20]

not a question of if but of when bytecode tool to generate the licens elibrary on the fly, using random permutations and injecting it everywhere into the bytecode with an open platform we have to accept a crack will happen [17]

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

alles hilft gegen lucky patcher auf den ersten blick, jedoch custom patches, welche Lucky Patcher anbietet[20], können es einfach umgehen, deswegen hilft nur reengineering schwerer zu machen viele piraten sind nicht mehr motiviert wenn es zu schwer ist every new layer of obfuscation/modifcation adds another level complexity

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

5.3 Future Work

- test

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

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

until lvl see major update custom improvements have to be done [20]

nicht mehr zu rettendes model, dex hat zu viele probleme, google bzw die andern anbieter müssen eine uber lösung liefern denn für den einzelnen entwickler so etwas zu ertellen ist nicht feasable, da einen mechanismus zu erstellen komplexer ist als die app itself se/tee muss es eine lösung geben sonst braucht man für verschiedene apps verschiedene se, gemeinsame kraft um die eine lösung zu verbessern und nicht lauter schweizer käse zu ahben

google hat schon sowas wie google vault

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

List of Figures

List of Tables

List of Code Snippets

Bibliography

[1] Alastair Beresford, et al. *All Vulnerabilities*. URL: http://androidvulnerabilities.org/all (visited on 01/24/2016).

[2] allatori. *Allatori Java Obfuscator*. URL: http://www.allatori.com/ (visited on 01/22/2016).

[3] allatori. *Documentation*. URL: http://www.allatori.com/doc.html (visited on 01/22/2016).

[4] allatori. *ProGuard*. URL: http://developer.android.com/tools/help/proguard.html (visited on 01/22/2016).

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

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

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

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

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

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

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

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

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

[14] D. Galpin. *Proguard, Android, and the Licensing Server.* URL: `http://android-developers.blogspot.de/2010/09/proguard-android-and-licensing-server.html` (visited on 01/22/2016).

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

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

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

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

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

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

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

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

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

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

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

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

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