TU WIEN Informatics

Analysis and Bypass of Android Application Anti-Reverse Engineering Mechanisms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Paul Kalauner
Matrikelnummer 11776818

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Raphael Kiefmann
Clemens Hlauschek

Wien, 28. August 2023

Unterschrift Verfasser

Unterschrift Betreuung

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Bibliothek, Your knowledge hub
TU WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Analysis and Bypass of Android Application Anti-Reverse Engineering Mechanisms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Paul Kalauner
Registration Number 11776818

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Raphael Kiefmann
Clemens Hlauschek

Vienna, 28th August, 2023

Signature Author

Signature Advisor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Analysis and Bypass of Android Application Anti-Reverse Engineering Mechanisms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Paul Kalauner
Registration Number 11776818

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Advisor: Thomas Grechenig

Wien, 28th August, 2023

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Paul Kalauner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 28. August 2023

Paul Kalauner

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First and foremost, I would like to thank Raphael Kiefmann and Clemens Hlauschek for providing valuable technical and formal feedback on this work.

Further, I wish to express my gratitude to my parents, friends, and girlfriend who supported me during the time of working on this thesis.

Finally, I want to thank my colleagues, especially Daniel Marth, who provided me with beneficial technical input.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Im Laufe der Zeit ist die Popularität von mobilen Applikationen deutlich gestiegen. Da die Nutzung von mobilen Applikationen die Verarbeitung sensibler Daten inkludiert, haben sich verschiedenste Empfehlungen und Spezifikationen bezüglich des Schutzes mobiler Applikationen mittels Anti-Reverse Engineering Mechanismen wie Obfuskierung und Root-Detektierung etabliert. Aufgrund der Erfordernis, mobile Applikationen zu schützen, wurden kommerzielle Anti-Reverse Engineering Tools, welche verschiedene Anti-Reverse Engineering Mechanismen implementieren, entwickelt. Allerdings haben Entwickler von Schadsoftware die Möglichkeit, Anti-Reverse Engineering Tools und Mechanismen zu nutzen, um die Analyse ihrer bösartigen Applikationen zu erschweren. Daher strebt diese Arbeit an, detaillierte Einblicke in die Funktionalität von verschiedenen durch Anti-Reverse Engineering Tools bereitgestellten Mechanismen zu erlangen, um Forschenden im Bereich der IT-Sicherheit eine effiziente Analyse von bösartigen mobilen Applikationen zu ermöglichen.

Diese Arbeit analysiert Anti-Reverse Engineering Mechanismen dreier Anti-Reverse Engineering Tools, indem eine Evaluations-Applikation, auf die die Mechanismen der Tools nacheinander angewandt wurden, statisch und dynamisch analysiert wird. Im engeren Sinne nutzt diese Arbeit verschiedene Reverse Engineering Techniken wie Dekompilierung und dynamische Code-Instrumentierung, um die Implementierungen von String- und Klassen-Verschlüsselung, TLS-Zertifikat-Pinning, und Root-Detektierungsmechanismen zu analysieren.

Basierend auf den in dieser Arbeit erhaltenen Analyseergebnissen werden Unterschiede zwischen den Implementierungen der analysierten Tools und Mechanismen diskutiert. Daraus resultierend werden im Rahmen dieser Arbeit Skripte, welche es erlauben den Großteil der analysierten Anti-Reverse Engineering Mechanismen dynamisch zu umgehen, entwickelt. Zusätzlich werden zwei Ansätze, die es ermöglichen automatisch zu detektieren, welches der drei analysierten Tools auf eine Applikation angewandt wurde, vorgestellt. Damit ist es möglich, die entsprechenden Skripte zur Umgehung der Mechanismen automatisiert auszuführen. Abschließend – basierend auf den vorgehenden Resultaten und Umgehungsstrategien – präsentiert diese Arbeit mögliche Ideen und Ansätze, um die analysierten Anti-Reverse Engineering Mechanismen und Tools zu verbessern.

Keywords: *Android, Mobile Sicherheit, Applikationsanalyse, Anti-Reverse Engineering*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

In the course of time, the popularity of mobile applications has increased drastically. As the usage of mobile applications includes the processing of sensitive data, various recommendations and specifications regarding the protection of mobile applications through anti-reverse engineering mechanisms, such as obfuscation and root detection, have been established. Due to the need for mobile application protection, commercial anti-reversing tools implementing various anti-reverse engineering mechanisms have emerged. However, malware developers might take advantage of anti-reversing tools and mechanisms in order to hinder analysis of their malicious applications. Therefore, this thesis aims to gain detailed insights into the functionality of various mechanisms provided by anti-reverse engineering tools in order to enable security researchers to analyse malicious mobile applications efficiently.

This thesis inspects anti-reverse engineering mechanisms of three anti-reverse engineering tools through statically and dynamically analysing an evaluation application, where the mechanisms provided by the tools have been applied to one after the other. More specifically, this work makes use of various reverse engineering techniques, such as decompilation and dynamic code instrumentation, in order to analyse the implementations of string as well as class encryption, TLS certificate pinning, and root detection mechanisms.

Based on the analysis results obtained in this work, implementation differences between the analysed tools and mechanisms are discussed. As a result, this work develops scripts for dynamically bypassing the majority of the analysed anti-reverse engineering mechanisms. In addition, this thesis introduces two approaches for automatically detecting which of the three analysed tools has been applied to an application, allowing to automatically execute the corresponding scripts for bypassing the applied mechanisms. Finally, building upon the previous findings and bypassing strategies, this work presents possible ideas and approaches for improving the analysed anti-reversing tools and mechanisms.

Keywords: *Android, mobile security, application analysis, anti-reverse engineering*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Problem Description 1
1.2 Goals 2
1.3 Methodological Approach 3
1.4 Structure 4

2 Related Work **5**
2.1 Android Application Security 5
2.2 Anti-Reverse Engineering Tools 5
2.3 Code Obfuscation 6
2.4 Code Encryption 6
2.5 String Encryption 7
2.6 TLS Certificate Pinning 7
2.7 Root Detection 8
2.8 Hooking and Debugging Detection 8
2.9 Emulation Detection 9

3 Background **11**
3.1 Android Fundamentals 11
3.2 Android Application Reverse Engineering Techniques 25
3.3 Android Application Anti-Reverse Engineering Mechanisms 29
3.4 Reverse Engineering Tools 37

4 Mobile Anti-Reverse Engineering Tools **45**
4.1 DexProtector 46
4.2 LIAPP 46
4.3 DashO 47

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5 Analysis of Mobile Anti-Reverse Engineering Mechanisms **49**
5.1 Analysis Approach and Setup 49
5.2 Analysis Procedure and Results 52
5.3 Main Differences Between Analysed Tools and Mechanisms 71

6 Bypass of Anti-Reverse Engineering Mechanisms **75**
6.1 DexProtector 75
6.2 LIAPP 78
6.3 DashO 79
6.4 Automatic Identification of Applied Anti-Reversing Tool 80

7 Possible Improvements of Anti-Reverse Engineering Mechanisms **85**
7.1 Possible Improvements 85
7.2 Expert Evaluation 89

8 Conclusion and Future Work **93**

A Appendix **95**
A.1 Expert Interview Guide 95

List of Figures **99**

List of Tables **101**

List of Listings **103**

Acronyms **105**

Bibliography **109**

Online References **121**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

This chapter introduces the problem, goals, methodological approach, and structure of this thesis.

1.1 Problem Description

Mobile applications have become more and more present over the years. Nowadays, a majority of online activities – such as browsing the web or using online banking – are accomplished using applications on mobile devices. From the security engineering perspective, mobile devices and applications represent a whole new potential attack vector, especially as mobile devices tend to store a lot of sensitive data [130].

Applications running on the most popular mobile operating system Android [233] are usually not compiled to machine code but to bytecode, which is later executed by the *Android Runtime (ART)* [44]. As bytecode is a higher-level representation and typically includes more meta information than machine code, Android applications tend to be easier to decompile than applications built for other operating systems.

To counteract possible reverse engineering attempts, various security and anti-reverse engineering mechanisms, as presented by several authors, such as Graux et al. [115], Haupert et al. [124], Sihag et al. [227], or Zhang et al. [265], were developed and improved subsequently. Developers often make use of *obfuscation*, the process of changing code in order to make it harder to understand without changing its intended functionality. In addition to obfuscation, several other anti-reverse engineering mechanisms, such as root detection [188] [234] or TLS certificate pinning [58] [61], can be employed to impede comprehending the logic and implementation of an application. Although anti-reversing mechanisms do not guarantee protection against all possible reverse engineering attempts, such mechanisms aim to increase the effort, time, and cost needed for reverse engineering up to a point where analysis is practically not attractive anymore.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As reverse engineering became a prominent problem in mobile application development, various recommendations and specifications regarding the usage of anti-reversing mechanisms have arisen. For example, the *Open Worldwide Application Security Project (OWASP)* [200] maintains the *OWASP Mobile Top 10* [199], which represents a list of the ten most common risks for mobile applications, including reverse engineering. Additionally, the OWASP maintains the *OWASP Mobile Application Security Verification Standard* [197] and *OWASP Mobile Application Security Testing Guide* [198], aiming to establish and test the compliance of security requirements for developing secure mobile applications. From these lists and recommendations containing common security risks, security requirements, and testing criteria, requirements for employing anti-reverse engineering mechanisms are derived.

Aiming to mitigate reverse engineering and to fulfil recommendations and/or potential requirements, several commercial anti-reverse engineering tools implementing various anti-reversing mechanisms have emerged. While some anti-reversing tools are business-to-business products or only available as part of complex frameworks, such as the mobile protection solution developed by Thales[1] [238], some of the more accessible and popular tools are *DexProtector* [145], *LIAPP* [165], and *DashO* [205].

Berlato and Ceccato [57], Haupert et al. [124], or Sihag et al. [227], for example, actively analyse anti-reverse engineering techniques for Android applications. However, to the best of our knowledge, specific implementation details of anti-reversing mechanisms provided by anti-reversing tools used in practice and potential approaches for bypassing the applied mechanisms are mostly unknown. Furthermore, the field of mobile application security is constantly evolving and therefore represents an arms race between attackers and defenders, thus requiring continuous research. Additionally, also malware authors have the possibility of taking advantage of anti-reversing tools and mechanisms to make analysis of their developed malware unattractive. Thus, it is in the interest of security analysts to have the possibility of efficiently analysing malicious applications despite anti-reverse engineering mechanisms being employed, for which a better understanding of current anti-reversing tools and mechanisms as well of their weaknesses is required.

1.2 Goals

Due to the specific implementations of anti-reversing mechanisms provided by current anti-reversing tools mostly being unknown, this thesis aims to fill the gap in research by getting a detailed insight into the functionality and weaknesses of anti-reverse engineering tools and mechanisms. This work provides a detailed analysis of the implementations of several mechanisms employed by the three mobile application anti-reverse engineering tools *DexProtector* [145], *LIAPP* [165], as well as *DashO* [205]. While these tools provide several anti-reverse engineering mechanisms, this work focuses on *class and string encryption*, *TLS certificate pinning*, and *root detection*. The mechanisms in focus are offered by several popular anti-reverse engineering tools [124] [257].

[1] Personal communication

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Further, based on results of the analysis of the anti-reversing mechanisms, this thesis develops procedures that allow analysts to bypass anti-reverse engineering mechanisms in order to analyse malicious applications more efficiently. This work also realises approaches for identifying which of the analysed tools was applied to an application, allowing to execute the established bypass approaches automatically. Additionally, this thesis establishes a list of possible approaches and ideas for improving the analysed mechanisms and tools.

More specifically, this thesis aims to answer the following research questions:

- **RQ1:** How do DexProtector, LIAPP, and DashO implement class and string encryption, TLS certificate pinning, and root detection mechanisms?
- **RQ2:** How can the mechanisms that have been analysed as part of **RQ1** be bypassed?
- **RQ3:** Based on the results of **RQ1** and **RQ2**, what are possible improvements of the analysed mechanisms in order to mitigate the identified possible bypassing strategies?

1.3 Methodological Approach

As a starting point, literature research is performed. In addition to fundamental theoretical aspects of Android and the relevant technologies, existing work related to Android security, class as well as string encryption, TLS certificate pinning, and root detection mechanisms is researched, as the literature could give some pointers on how anti-reversing mechanisms are typically implemented. Researching existing work regarding bypassing anti-reverse engineering mechanisms is also part of this phase, as the following practical analysis part could benefit from existing bypassing strategies.

After establishing a proper understanding of the Android platform and (anti-)reverse engineering tools and mechanisms, we analyse how DexProtector [145], LIAPP [165], and DashO [205] implement string and class encryption, TLS certificate pinning, and root detection mechanisms. As all three tools this thesis focuses on are closed source tools, we perform several steps for each of the investigated tools: (1) Applying the tool to an evaluation Android application, (2) statically analysing the resulting protected application, and (3) dynamically analysing the application during runtime.

First, we develop a minimal evaluation Android application, allowing to analyse the applied mechanisms without additional overhead. For the purposes of this research the evaluation application must only include a basic HTTPS request with the corresponding pinned TLS certificate. Afterwards, for each of the analysed mechanisms, we create a build of the developed evaluation application, where the corresponding mechanism that is currently analysed has been applied to.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Second, we employ *static analysis* (e.g. by using decompilation tools) aiming to gain an understanding of the functionality and code structure of the anti-reversing tools and mechanisms. In the context of this work, we check the produced evaluation application after the tool with the corresponding anti-reversing mechanism has been applied to it, which potentially reveals details on how the mechanism and tool are operating.

Third, we employ *dynamic analysis* with a focus on code instrumentation by executing the protected evaluation application while observing and modifying its behaviour. For analysing anti-reversing mechanisms, we assume some of the executed operations of the protected application and check the made assumptions through executing the application. For example, root detection mechanisms commonly check existing files that indicate a rooted device [234]. In this case, employing code instrumentation and intercepting functions used to open files allows verifying this behaviour.

Based on the results of the previous analysis phase, we develop approaches for automatically bypassing the anti-reversing mechanisms provided by DexProtector, LIAPP, and DashO. In practice, the anti-reversing tool and mechanisms that have been applied to a given application are usually not known up front. Therefore, in order to provide an automatic way to bypass the anti-reversing mechanisms of a given application, we additionally develop an approach to identify which of the anti-reversing tools this work focuses on was applied to an application.

Lastly, based on the results of the previous analysis and developed bypassing strategies, we conceptualise possible improvements of anti-reverse engineering mechanisms and the analysed tools to help mobile application developers protecting their applications. The presented improvements are evaluated by means of interviews with several experts in the field of IT security.

1.4 Structure

The remainder of this work is structured as follows: Chapter 2 provides an overview of related literature. Chapter 3 provides the necessary theoretical foundation of the Android operating system, reverse engineering techniques, anti-reverse engineering mechanisms, and reverse engineering tools, as the following chapters build upon this knowledge. Next, chapter 4 presents the anti-reverse engineering tools this work focuses on. The main part of this thesis consists of chapters 5 to 7. Firstly, chapter 5 concerns **RQ1** and analyses several anti-reversing mechanisms provided by anti-reversing tools through employing various reverse engineering tools and techniques. Secondly, **RQ2** is addressed in chapter 6, which develops strategies for bypassing the analysed mechanisms based on the previous analysis results. Thirdly, in chapter 7, we present and evaluate possible approaches for improving the investigated anti-reverse engineering tools, thus answering **RQ3**. Finally, we conclude the work in chapter 8.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Related Work

The following literature is either tangential or directly relevant to our work.

2.1 Android Application Security

Makan and Alexander-Bown [171] describe reverse engineering approaches for Android applications and demonstrate simple emulator/root detection and obfuscation mechanisms.

Gunasekera [118] focuses on reverse-engineering Android applications as well and additionally presents several reverse engineering tools, such as Frida [95].

He et al. [125] focus on Android malware and argue that, compared to other mobile platforms, Android is more prone to attacks due to its openness. This openness does not only lead to an increased malware risk, it also allows employing reverse engineering techniques more easily compared to other platforms.

Enck et al. [89], Vidas et al. [248], Shabtai et al. [225] as well as Xu et al. [260] describe how Android tries to establish security through its built-in security features, such as the permission and sandboxing system.

2.2 Anti-Reverse Engineering Tools

Cho et al. [64] present "DexMonitor", an approach that aims to print all executed bytecode of an application by placing hooks in the Dalvik VM where Dalvik instructions are about to be executed. For evaluating their approach, the authors use three applications protected with obfuscation and tamper detection. However, there is no information about the tools and protection mechanisms that have been employed for protecting the evaluation applications. Nevertheless, the authors outline a selection of anti-reversing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

tools, including DexProtector [145]. The authors also provide a feature overview table of the tools.

Lim and Yi [150] analyse the structure of two anti-reverse engineering tools, with one of them being DexProtector. However, as the paper was written in 2016, implementation details have changed since then. Hence, the analysis results are most likely out of date.

Haupert et al. [124] provide an overview of available *runtime application self-protection (RASP)*/anti-reversing tools and their features, such as code obfuscation or root detection. Further, the authors analyse the popular tool *PromonShield* [209] and present static and dynamic approaches for bypassing its offered security measures.

Sihag et al. [227] discuss and compare several Android application hardening/anti-reversing techniques. Additionally, the authors compare various Android application hardening tools by their offered hardening techniques.

2.3 Code Obfuscation

With the intention of impeding reverse engineering and protecting intellectual property, code obfuscation was firstly discussed in an academic context by Collberg et al. [72]. The authors distinguish between layout obfuscation (scrambling identifiers, removing comments, etc.), data obfuscation (reordering methods, splitting and merging arrays, etc.), and control flow obfuscation (inlining methods, reordering statements, extending loop conditions, etc.).

Graux et al. [115] as well as Faruki et al. [92] discuss different kinds of obfuscation (identifier renaming, control flow obfuscation, obfuscation through reflection, etc.) and available tools applicable to Android applications.

Guo et al. [120] investigate different strategies for obfuscating Android applications and discuss several approaches for analysing and deobfuscating obfuscated applications. These approaches include techniques based on machine learning and dynamic analysis.

Furthermore, several approaches for detecting obfuscated Android applications have been proposed. Primarily, researchers, e.g. Mirzaei et al. [182], Jiang et al. [132], or Conti et al. [77], leverage machine learning-based techniques.

2.4 Code Encryption

One of the first occurrences of code encryption were *packers*, which were primarily leveraged by malware developers in order to conceal malicious programs. Guo et al. [119] discuss the underlying mechanisms of packers and potential solutions to unpack packed binaries intending to enable malware analysis. Packers transform a binary program into another form, leading to a different appearance than the original, thus evading signature-based malware detection. During this process, packers might also employ code

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

encryption and/or compression to make manual unpacking and static analysis even more cumbersome.

In the context of Android, the principles of code encryption are discussed in the work of Graux et al. [115]. On Android, the most popular form of encrypting code is *class encryption*, the process of encrypting entire classes. Thus, viewing the source code of the encrypted classes by using usual decompilation tools is prevented.

Geethanjali et al. [100] developed a tool for encrypting and dynamically loading and decrypting code at an Android application's runtime.

Zhang et al. [266] propose a technique for extracting hidden code from Android applications. The authors describe the basic idea and process of code hiding/encryption and discuss extracting strategies.

2.5 String Encryption

String encryption is used to hide the content of string constants by encrypting them using cryptographic operations. This way, the encrypted strings stay hidden, even if the code is decompiled. As a result, string encryption helps hiding sensitive data, such as URLs of command and control servers.

String encryption is a type of the more generalised technique *string obfuscation*, the technique of hiding strings, regardless whether this is achieved using cryptographic operations or other methods. Glanz et al. [107] provide empirical evidence that string obfuscation is widely used in malicious as well as benign Android applications. Furthermore, the authors propose "StringHound", an approach for automatically deobfuscating strings in Java bytecode.

Yoo et al. [261] present a method to decrypt encrypted strings and discuss the deobfuscation tool "dex-oracle" [101].

2.6 TLS Certificate Pinning

Sierra and Ramirez [226] research how effective common reverse engineering techniques are against TLS certificate pinning and give a general introduction to TLS certificate pinning. This mechanism is used to restrict an application's secure connection to specific certificates by providing the trusted certificate(s) or its/their public key(s). Doing so will prevent any other connections using certificates that are not pinned, preventing man-in-the-middle attacks. Malware authors employ TLS certificate pinning to conceal the information exchange of their malicious applications. Sierra and Ramirez present three methods to bypass TLS certificate pinning: Reverse engineering and modifying the source code, hooking during runtime, and dynamic library manipulation. In addition, the authors discuss mitigation techniques, such as emulator detection or code obfuscation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Ramirez et al. [215] present multiple mechanisms, such as root or debugging detection, to protect against methods aiming to bypass TLS certificate pinning.

Fahl et al. [91] analyse the potential risk of flawed TLS certificate pinning. The tool "MalloDroid", which the authors developed as part of their work, helps detecting potential vulnerabilities to protect against man-in-the-middle attacks within Android applications.

2.7 Root Detection

Nguyen-Vu et al. [188] present the fundamentals of rooting Android devices. As the name suggests, rooting allows obtaining root access on Android. Due to the higher privileges, rooting often represents a security risk. Applications that deal with sensitive data often use root detection to prevent the execution in case the device is rooted. Malware authors use root detection to prevent the analysis of their malicious applications as, e.g. dynamic code instrumentation tools rely on the device being rooted. Furthermore, the authors discuss different methods for detecting a rooted Android device. In addition, they apply various evasion techniques and analyse their effectiveness.

Sun et al. [234] present further methods for detecting a rooted device. For each of the presented methods, the authors propose a corresponding evasion technique.

Ibrahim et al. [128] analyse several applications that are using the *SafetyNet Attestation API* [35]. The SafetyNet Attestation API, nowadays replaced by the *Play Integrity API* [34], is an anti-abuse API offered by Google aiming to ensure that a genuine Android device (i.e. amongst other checks, it is verified that the device is not rooted) is used for running an application. The authors conclude that out of the analysed applications, none of them is using the API correctly, resulting in the possibility of attackers being able to bypass the checks.

2.8 Hooking and Debugging Detection

Berlato and Ceccato [57] conduct a large-scale study about the usage of anti-debugging and anti-tampering techniques in popular Android applications. Their results suggest that about two out of three applications leverage such techniques. However, the ratio of Java to native implementations of such mechanisms is 99 to 1, implying that most developers rely on fundamental and known approaches to detect debugging and tampering.

Szczepanik et al. [237] present an approach to detect the usage of the common hooking framework *Xposed* [258]. Their approach relies on analysing stack traces of the executed application and comparing them to the expected executed statements and program flow using machine learning approaches. The expected program flow is retrieved through analysing the Dalvik bytecode contained in the DEX file(s) of an application's APK.

Wan et al. [253] introduce an approach for detecting the usage of debugging or hooking tools through "check points" (memory addresses). In case the control flow of an application

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

is modified through debugging or hooking, the check points will usually be tampered with. In order to detect tampered check points, the authors' approach collects and stores various memory addresses, such as the addresses of methods stored in the virtual table, as a preliminary step. During runtime, the approach frequently checks whether the actual addresses match with the stored ones. If not, the application is terminated.

2.9 Emulation Detection

Vidas and Christin [247] present several approaches for detecting emulated devices. A fundamental approach consists of retrieving system properties that hold different values on physical and emulated devices. Further, the authors discuss the possibility of detecting emulators based on the differences in behaviour, performance, and components.

Petsas et al. [202] discuss different heuristics for detecting emulators: Static, e.g. system properties, dynamic, e.g. sensor information, and hypervisor heuristics. Hypervisor heuristics aim to detect emulators by analysing the behaviour of the underlying execution environment.

Choi et al. [67] present *EmuID*, an approach for detecting ARM emulators through a specific behaviour regarding caching on ARM architectures. EmuID uses a crafted code that causes a specific cache behaviour on native environments. However, emulated environments are usually not able to reproduce this exact behaviour, leading to the possibility of detecting emulators.

Lin et al. [151] introduce an emulator detection approach consisting of three layers to detect the respective characteristics, namely the OS, hardware, and hypervisor layers. The authors evaluate their approach on various common emulators.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 3

Background

The following chapter introduces background information related to the Android mobile operating system, reverse engineering techniques, anti-reverse engineering mechanisms, and reverse engineering tools.

3.1 Android Fundamentals

Android [2] is a mobile operating system, developed and maintained by Google. At the time of writing, Android holds ~70% market share amongst mobile operating systems [233].

This section provides the necessary Android-related background information for this work. This includes knowledge about the Android system's architecture, structure as well as components of Android applications, and the security model of Android.

3.1.1 Architecture

Android is composed of several layers, forming a stack, as shown in Figure 3.1. The following sections describe each layer/component of the Android platform.

Kernel

The Linux kernel represents the foundation of the operating system. As common in other Unix operating systems, the Android kernel provides several drivers for networking, file-system access, process and memory management, etc. [87]. Android uses a modified and feature-enhanced Linux kernel.

Hardware Abstraction Layer

On top of the kernel, the *hardware abstraction layer (HAL)* is located. The HAL provides interfaces that expose hardware capabilities to upper layers of the stack and allows

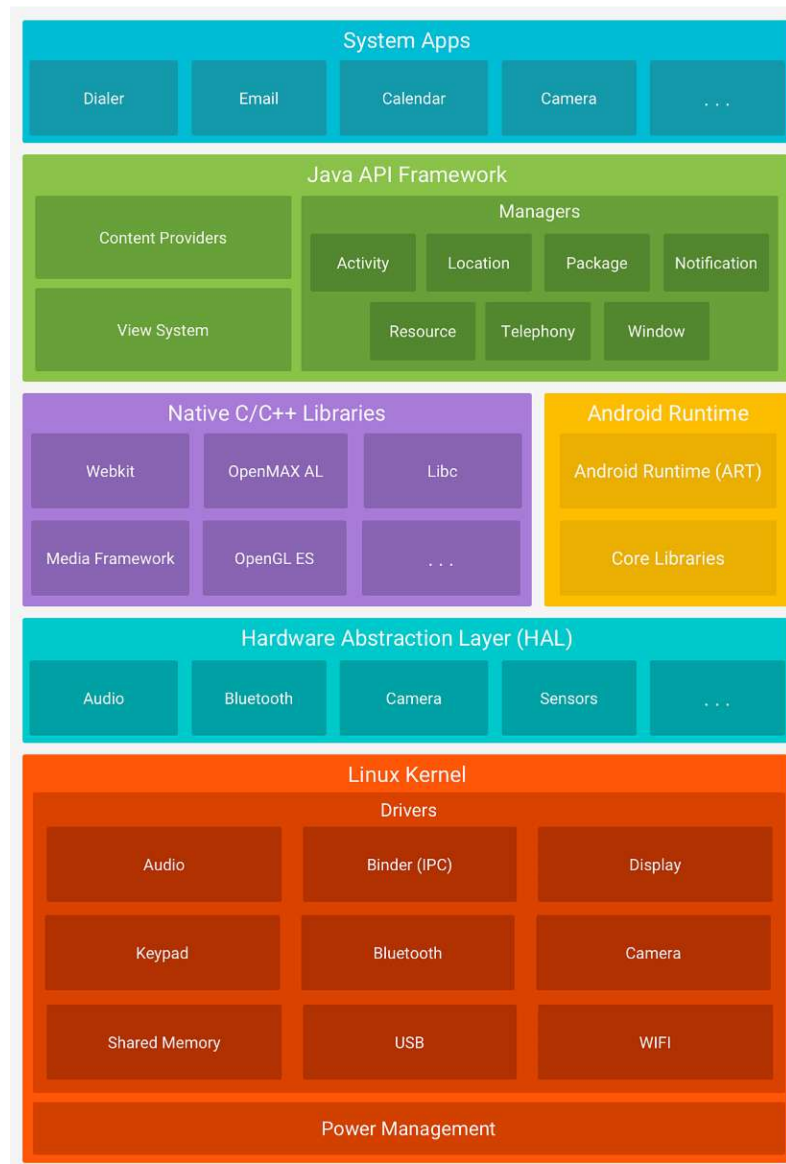


Figure 3.1: Android architecture [33].

porting Android to a wide range of different devices [183] [33]. It consists of multiple modules, with each module implementing an interface for a particular type of hardware component, such as Bluetooth [33].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Native User Space

The native user space layer consists of several native (C/C++) libraries, e.g. *bionic* [109], the Android-specific implementation of *libc* [155], used by the Android operating system as well as *core system services* [87].

Core system services are needed to setup the underlying OS environment [83]. These services include `init` that is the first user space process launched by the kernel on Linux and therefore also Android systems. `init` mounts several directories and executes a series of commands ("startup scripts") to setup the user space environment, e.g. starting services [83] [187].

Part of the Android-specific `init` process is the *property service*, which provides a mapping of key-value pairs. These mappings are shared between all processes and include properties such as network interface configuration or device information [83] [75].

Furthermore, `init` starts several daemons, including the volume daemon `vold`, which is responsible for mounting and unmounting file systems [83].

After starting the daemons, `init` launches the *zygote* process [183]. zygote launches the first instance of the *Dalvik Virtual Machine (DVM) / Android Runtime (ART)* and preloads core classes and resources. When a new application is started, it creates a new, separate instance of the DVM/ART, inheriting from the initially created instance with its preloaded classes and resources [183] [83]. Thus, new VM instances can be created in an efficient way. Another responsibility of zygote is starting the `system_server` process that in turn starts the system services.

Dalvik VM / Android Runtime

Android applications are commonly written in the Java or Kotlin language and compiled into bytecode as an intermediary step before they are compiled into the *DEX (Dalvik Executable)* format stored in `.dex` files within packaged Android applications.

A virtual machine (VM) – similar to the Java Virtual Machine (JVM) – is needed to interpret and execute the bytecode, which the *Dalvik Virtual Machine (DVM)* was used for originally. The main difference between the DVM and JVM is that Dalvik is register-based as opposed to the JVM that uses a stack-based approach. Furthermore, DVM was designed to run efficiently on devices with limited memory and processing power. As a result, it is more lightweight than the JVM and allows multiple instances to be run at the same time [63].

The DVM is using a *just-in-time (JIT)* compilation approach, meaning that the compilation of bytecode to machine code is delayed until it is actually needed during runtime. This approach leads to worse performance and the need of additional resources during the execution of an application [63].

To circumvent this loss in performance, Android 5.0 introduced the *Android Runtime (ART)*, which replaces the Dalvik VM [63] [260]. As opposed to the Dalvik VM, ART

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

introduced *ahead-of-time (AOT)* compilation during the installation of applications. AOT compilation converts bytecode to machine code, which is stored in `.oat` binary files. Consequently, the installation process became more time consuming. Furthermore, the size of applications increases noticeably due to the majority of application code being compiled to machine code when using AOT compilation. However, the AOT approach increases application load times and responsiveness [63].

To leverage the advantages of both DVM and ART, Android 7.0 introduced a hybrid combination of AOT and JIT compilation to ART. This hybrid approach, which is specified in the Android Open Source Project documentation [45], eliminates the need of bytecode having to be compiled to machine code during an application's installation. Instead, during the first few application launches code is interpreted with frequently used methods being JIT compiled. Additionally, the first few runs are profiled and frequently used code is determined. At a later point of time (e.g. when the device is idle and/or charging), the frequently used code detected previously is AOT compiled and permanently stored on the device. Therefore, future application launches can take advantage of precompiled code of frequently needed code segments.

Java Core Libraries

Android provides an own set of Java core libraries that were derived from the Apache Harmony project [50], which aimed to develop an open-source Java implementation. Over time, the implementation of Android's runtime libraries diverged more and more from Apache Harmony [87]. The runtime libraries can be accessed from applications as well as system services.

System Services

System services implement fundamental Android features, e.g. display/touch screen support and telephony. Most services are implemented in Java, although some fundamental ones are written in C/C++ [87]. Communication between system services is established using *Binder*, described in section 3.1.3.

Java API Framework

The Java API Framework consists of Java libraries that are not part of the standard Java runtime. The framework includes classes for fundamental components of Android applications. Furthermore, classes that enable interactivity with hardware and system services ("managers") are part of the framework [87].

Applications

On the top of the stack, Android applications are located. It is differentiated between *system* and *user-installed* applications. System applications are pre-installed and include

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

browsers or calendar applications. In contrast, user-installed applications are installed by the users themselves.

3.1.2 Applications

Android differentiates between pre-installed (system) and user-installed applications. System applications are included in the OS image and usually installed in the read-only mounted `/system` partition [87]. Hence, these applications cannot be uninstalled or modified by normal users and may have elevated privileges compared to user-installed applications. In contrast, user-installed applications are installed from Android's application market *Google Play* [112] or *sideloaded*, i.e. installed from sources other than the Google Play Store, e.g. a web download. Applications installed by the user reside in the `/data/app` directory [83].

Android applications are packaged in *Android application package (APK)* files which comprise the files necessary for a fully installation-ready application. An APK is a ZIP-archive with the structure shown in Figure 3.2. More precisely, it contains the following files and directories [183] [63]:

- `AndroidManifest.xml`: The Android manifest contains meta information and configuration of application components.
- `classes.dex`: This file contains the Dalvik bytecode of an application that will be run through the DVM/ART. Note that nowadays, *multidex* is enabled per default, leading to an application's code potentially being split into multiple DEX files [24].
- `resources.arsc`: Pre-compiled resources, e.g. binary XML files to reduce processing costs, are stored in `resources.arsc`.
- `META-INF/`: The `META-INF` directory commonly includes meta data of the application, such as signature information.
- `assets/`: This directory contains resources that are packaged as-is. Developers access these files in a typical file system-like manner.
- `res/`: This directory contains pre-processed – but not pre-compiled – resources, such as layouts or images. Developers can access resources stored here using unique resource identifiers.
- `lib/`: Native libraries used by the application are typically stored as *shared objects* (`.so` files) inside the `lib` directory, which contains a subdirectory for each architecture.

In the following, we describe the main components of an Android application.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

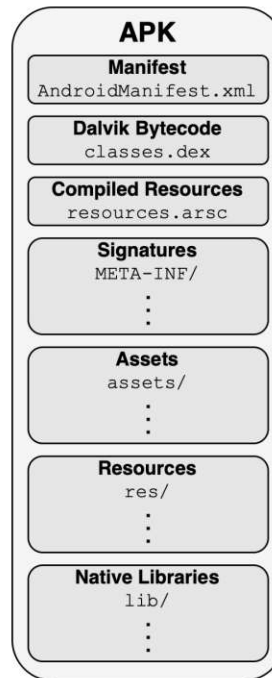


Figure 3.2: Structure of an APK file.

Android Manifest

The `AndroidManifest.xml` file contains various kinds of information about the application, such as its unique package name (e.g. `com.example.app`) and device compatibility [83] [10]. Other essential parts of the manifest are the following.

First, the components of the applications have to be declared. These include *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*.

Furthermore, the permissions (e.g. to access location data) needed by the application have to be declared. Additionally, for each activity, service, and broadcast receiver, the manifest declares which *intents* (see section 3.1.2) can be handled [10]. These declarations are called *intent filters*.

Intents

Intents are asynchronous messaging objects used for requesting actions from another application component [28].

Android distinguishes between two types of intents [28]:

- *Explicit intents* specify which application or component will handle the intent [28]. They are often used to start an application component within the same application.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To do so, the class name of the component has to be known and specified. For example, explicit intents are often used to switch to another activity.

- *Implicit intents* do not declare a specific application or component. Instead, they declare an action to perform, e.g. opening an URL or sending an email. The Android system then chooses a suitable application/component to perform the action. Usually, this decision is delegated to the user by showing an application picker. For an application component being selectable for a specific action, a corresponding *intent filter* must be declared in the Android manifest by the developer [63].

Activities

As stated by the Android developer guide regarding application components [13], activities are entry points for interacting with the user. To name some examples, an activity could represent a screen showing a list of messages and another one could provide the functionality to compose a new message. While multiple activities usually work together, they are independent of others. As a result, an application can start an arbitrary activity of another application if it allows it.

Nowadays, applications are often developed using a single activity and multiple *fragments* for different views [116]. A fragment represents a reusable part of the UI/sub-activity which is displayed inside the layout of an activity [26] [116].

Services

Services are components running in the background without an user interface [83]. They are used for performing long-running operations – such as playing music or downloading a file – and usually keep running even if the application that started the service is moved to the background or another application is opened [63].

There are two types of services [13] [63]:

- *Started services* typically do not need to communicate back to the application that started them and are kept running until their work is completed. Regular background services are not directly visible to the user and may be stopped by the system if more resources for other tasks are needed. In contrast, services that should not be stopped in any circumstances are required to indicate that they are running by showing a notification to the user ("foreground services").

- *Bound services* are started because another application or the system wants to make use of the service. Consequently, a bound service communicates the results back to the calling application. However, as bound services are *bound* to the application that started them, they are killed as soon as the calling application is terminated.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Broadcast Receivers

Broadcast receivers respond to system-wide events ("broadcasts"), such as an incoming SMS message [87]. If an application registers for a specific event using a broadcast receiver, the system is able to deliver events to the application even if it is not running at the moment the event is triggered [13]. Thus, the receiving application can specify code that should be executed as soon as the event is received.

Content Providers

Content providers manage access to an application's data – either stored by itself or other applications – and provide a way to share data with other applications [18]. Therefore, if an application needs to share data, it may declare a content provider that exposes the data to other applications. The data exposed by content providers typically originates from an SQLite database or file system path [83].

Furthermore, content providers provide an abstraction to the specific data storage implementation. This way, application developers can exchange the storage implementation accessed by the content provider without affecting other applications that access the data [18]. An example for this procedure is illustrated in Figure 3.3, where an SQLite database is migrated to another storage system. However, due to the content provider's abstraction, this migration is performed transparently to the accessing applications.

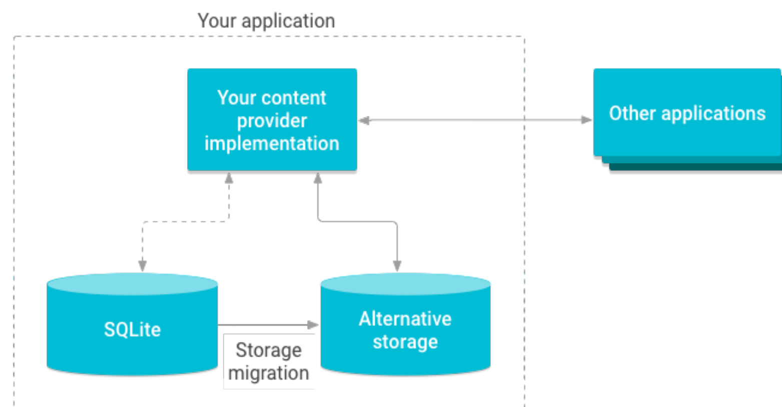


Figure 3.3: Android content provider overview and storage migration [18].

Native Libraries

Although Android applications are mostly written in Java or Kotlin, Android allows embedding native code (using languages such as C/C++) via the *Android Native Development Kit (NDK)* [8]. The NDK builds native libraries from native code, which are

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

then included in the `lib/` directory of the final APK. However, as native libraries are compiled for a specific target processor architecture, it is required that the APK contains several native libraries – one for each architecture that the application should run on.

While bytecode (compiled from Java/Kotlin) is executed by the ART, machine code (compiled from native code) is directly executed by the processor of the mobile device [228]. Therefore, for bytecode being able to interact with native code and vice-versa, the *Java Native Interface (JNI)* [194] is needed [29]. JNI allows native code to access fields and invoke methods defined in the Java/Kotlin code. At the same time, JNI allows Java/Kotlin code to invoke methods implemented in the native part of the application, which results in a two-way communication between these components [164]. Figure 3.4 illustrates the described relationship and interaction between Java/Kotlin code, native code and ART.

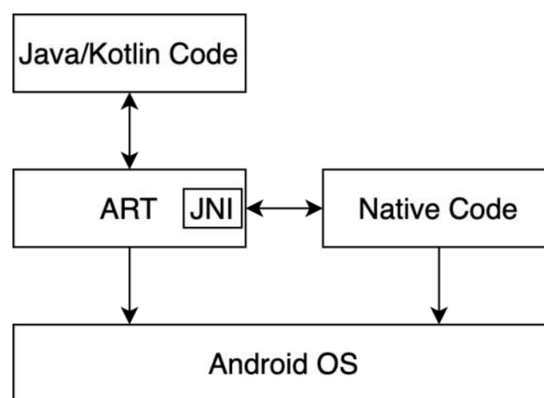


Figure 3.4: Interaction between Java/Kotlin-, native code, and ART, adapted from [164].

Using the NDK may improve the performance of applications, as native code is directly compiled to machine code, thus avoiding code execution on a virtual machine [142]. Especially computationally intensive tasks commonly benefit from an implementation in native code.

In addition, reverse engineering program logic implemented in native code is generally more cumbersome, as machine code preserves significantly less information from the original source code compared to bytecode [88] [121]. Therefore, implementing sensible logic in native code can help protecting intellectual property, although it is still possible to extract implementation details with enough effort and resources.

Furthermore, using the NDK allows developers to reuse existing libraries. This also enables developers to port their logic written in native code to other platforms as well.

3.1.3 Security Model

Security features of Android are deeply embedded into the operating system and design of the platform itself [173] [180] [183]. Most security mechanisms are provided by the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Linux kernel that represents the foundation of Android OS [228]. On top of the security features provided by the kernel, Android implements additional mechanisms, such as application-specific permission management.

The remaining part of this section introduces the major components of Android's security model.

Sandboxing

One of Android's core security principles is *application sandboxing* that ensures that applications cannot access data or memory of other applications [228].

The fundamental mechanism behind Android's sandboxing implementation is *discretionary access control (DAC)* of typical Linux desktop systems with physical users [260] [93]. As Linux is a multi-user operating system, the kernel isolates user resources and processes from one another [87]. Linux establishes this isolation by assigning each user a *user ID (UID)*. In addition, users can be added to groups that are identified by *group IDs (GIDs)*. Each resource, e.g. a file, is assigned a UID of the owner, who may alter the permissions of the resource [228]. On Android, UIDs are not assigned to physical users, as Android was originally designed for smartphones that are usually single-user devices [87]. Instead, Android automatically assigns a unique UID to each application at installation [87] [138]. The installed application is then run in an own process under this UID. In addition, each application has its own data directory with read and write permissions only granted to its UID. This way, no other application has permissions to read and write its data [87]. As a result, Android applications are *sandboxed* (isolated) at file and process level [87]. Sandboxing on process level is achieved through running each application in a dedicated process. File level sandboxing is a result of each application having its own isolated data directory.

Additionally, Android 4.2 introduced multi-user support, with a unique user ID – independently from the Linux UID – and a dedicated data directory, containing user-specific settings, being assigned to each user [87]. As the Linux kernel only supports a single numerical range for UIDs, applications installed for each user are assigned a new effective UID, also called *Android ID (AID)*, in order to still be able to distinguish applications installed for each user and guarantee application sandboxing [87] [173]. The AID is a composition of the user's ID and the application's UID [87]. More precisely, device users are separated through a large offset and UIDs for applications installed per user are assigned in a defined range [173]. Therefore, the effective UIDs differ, even if the same application is installed by different users. As a result, each application instance runs in its own sandbox [87].

Nonetheless, applications can also be installed using the same user ID, forming an exception to this principle of strict isolation. Applications with shared UIDs can share files and run in the same process [87]. Shared UIDs are mostly used by system applications and not recommended for third party applications. Nevertheless, if third party applications are signed with the same key (see section 3.1.3) and the corresponding attribute is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

included in the Android manifest file, third party applications are installed using the same UID as well [87].

Permissions

Due to the sandboxing of Android applications, they are limited in terms of accessing files and resources of the device [87]. More access rights – called *permissions* can be granted to applications, which allows access to several resources, such as internet connectivity. Applications have to declare the needed permissions inside the Android manifest [264].

Permissions are enforced at different levels of the Android stack. For example, access to application files is enforced by the Linux kernel, as applications' unique UIDs and GIDs are only given access to their corresponding data directories on the file system [83]. Similarly, if an application is permitted to access the network, the application's UID is added as a member of the `inet` group and therefore granting the application the ability to open network sockets [83].

Android differentiates between different types of permissions [32]:

- *Install-time permissions* give limited access to data and allow an application to perform actions that minimally affect other applications or the Android OS. Permissions of this type are requested before the user installs an application. There are two subtypes of install-time permissions:
  - *Normal permissions* allow limited access with minimal risk to data and actions beyond the sandbox of applications. For example, the ability to access the network is a normal permission.
  - *Signature permissions* can be used to share resources between multiple applications [227]. They are only granted if the application that declares the signature permission and the application that defines it are signed by the same developer certificate (see section 3.1.3) [56]. Signature permissions are also often used by system applications to change device settings.
- *Runtime permissions* (or *dangerous permissions*) allow applications to perform actions that affect other applications or the Android OS more significantly. Additionally, access to restricted data (e.g. private user data) may be granted. As the name implies, runtime permissions have to be requested during runtime of an application. For example, the ability to access the device's camera is a runtime/dangerous permission.
- *Special permissions* can only be defined by the original equipment manufacturer (OEM) and include powerful application operations, such as picture-in-picture mode or access to notifications. These permissions have to be granted manually in the system settings.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Binder

Binder is an inter-process communication (IPC) mechanism based on *OpenBinder* [201] and represents the central messaging channel on Android, providing application communication with the system, services, and each other [70]. Although Linux has built-in IPC mechanisms, Android uses Binder to remove the overhead and complexity of the standard IPC facilities. IPC is necessary as – on a Unix-like system – a process cannot access other processes directly [87]. Instead, the kernel has control over all processes and can expose an interface that enables IPC [87]. On Android, this interface is the `/dev/binder` device, implemented by the Binder kernel driver [87]. Figure 3.5 illustrates the communication between two processes using Binder. Through Binder's abstraction, two processes seem to be able to communicate with each other directly. In reality, all IPC calls take place via the Binder driver [87]. The driver performs the kernel-level tasks necessary for IPC, such as copying data from one process to another [53]. The kernel driver also adds the process ID and Android ID of the calling process to the transaction data [87]. This way, the called process can inspect the PID and Android ID in order to control access to resources. The second key component of Binder is the `libbinder.so` library ("Binder framework"), which is loaded into most processes [53]. The Binder framework is responsible for wrapping and unwrapping objects into simplified objects ("Parcels") and uses the `ioctl` [154] system call used to control a device via its driver to transfer the relevant data to the kernel/Binder driver [53] [78].

In addition to establishing the communication between system services, Binder is also used as a low-level foundation for higher-level IPC abstractions, such as Intents [87] [175]. Furthermore, Binder enables the communication for bound services. In order to allow clients (components, such as activities) to bind to the service, Android provides the `IBinder` interface, which developers can use to specify how clients can communicate with the service [14].

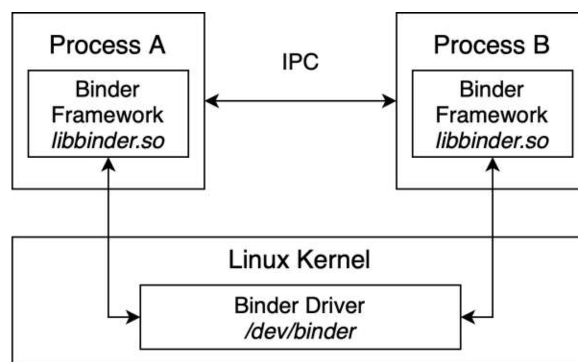


Figure 3.5: Simplified illustration of IPC via Android Binder.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Application Signing

The Android operating system requires Android applications to be signed by their developers in order to be executed [87]. While only the developer holds the private key of the certificate to sign the application, the public key is visible to everyone to verify the signature [254]. The use of a certificate authority to sign the certificate is not required, as self-signed certificates are sufficient [118].

Android uses the signing mechanism to ensure that updates for an application were created from the same, original author and not tampered with by a third-party. This verification is carried out by comparing the certificate of the currently installed application with the one of the updated application at installation time [87]. Furthermore, developers might sign two ore more applications with the same key in order to use a shared UID across multiple applications, allowing multiple applications to share files or to run in the same process [87]. In addition, application signing also allows the Android system to distinguish system applications from other applications, as the former are signed with the same certificate as the Android firmware. However, application signing on Android does neither guarantee that the signed code originates from a trusted developer nor that it is safe to run [87] [136].

Since August 2021, Google requires applications that are published in the official Play Store [112] to be uploaded using the *Android App Bundle (AAB)* format [5]. Applications that are distributed using APKs (e.g. for uploading them to alternative stores), still require the developer to sign the APK directly [37].

The AAB publishing format includes the complete compiled code and resources of an application, but is not distributed to end users as-is [37]. The introduction of AABs also required some changes to the application signing process, which is illustrated in Figure 3.6.

First, AABs must be signed using an *upload key* which is provided to Google and can be replaced with a newly generated one in case it gets compromised or lost [37]. After uploading the signed application bundle, Google builds several APKs tailored to specific device properties, such as display resolutions or processor architectures [140]. These APKs are signed with the *app signing key* – a key other than the upload key, which never changes [37]. Building different APKs for different devices results in smaller APKs, as only the needed resources are packaged [140].

SELinux

Android relies on Linux' discretionary access control (DAC) to enforce sandboxing and permissions. With DAC, a program runs with the permissions of the user who executes it, meaning that the program could perform all actions that the user is allowed to [174]. Although Android prevents applications from misusing granted permissions by assigning a unique UID to each application, two main weaknesses of DAC remain.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

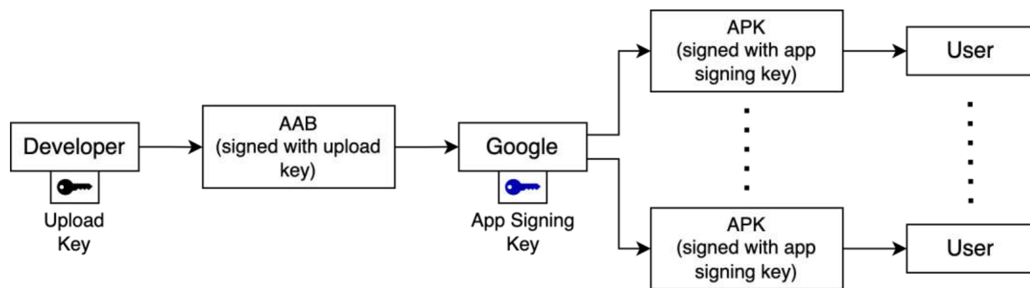


Figure 3.6: Application signing process using AAB, adapted from [37].

First, DAC allows users with permissions to a specific resource to change the permissions of said resource [83]. This could lead to exposing a potentially private resource to the public, intentionally or due to a programming error [83]. Second, DAC lacks the possibility to define permissions in a fine-grained way [230]. This is especially relevant for system processes that run as the root user [83] [174].

To overcome these shortcomings, Android integrated an Android-specific implementation of the *mandatory access control (MAC)* mechanism *SELinux*, short for *Security Enhanced Linux* [230]. MAC enforces a system-wide, fine-grained security policy that can only be changed by the system, not by unprivileged users/applications [87]. This security policy explicitly specifies which actions are allowed and is primarily enforced by the kernel [173]. As a result, even if users have access to a specific resource, they cannot modify the policy, e.g. in order to grant others access to the resource [87] [174].

This MAC model is based on three main concepts [83] [174]:

- *Subjects* are active actors that perform actions on objects. Usually, subjects are running processes.
- *Objects* are resources, such as directories, files, and processes.
- *Actions* are performed upon objects by subjects. Actions include operations such as reading, writing and executing.

These three concepts are the foundation of the core operation performed by SELinux – checking whether a certain subject is allowed to perform a certain action on a given object [174]. MAC does not replace, but supplements DAC [187]. DAC rules are checked first and if (and only if) access is allowed, the SELinux/MAC policies are checked afterwards [187].

Furthermore, SELinux has three modes of operations, which can be changed using the `setenforce` command [87]:

- *Disabled*: No policies are loaded, fallback to DAC security enforcement.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- *Permissive*: Policies are loaded and checked. However, violations of policies, i.e. access denials, are only logged and not enforced.
- *Enforcing:* Used on Android per default and can only be changed with root privileges. Policies are loaded and enforced. In addition, violations are logged.

3.2 Android Application Reverse Engineering Techniques

While the Android system provides a sophisticated security model, which provides fundamental security features, such as the isolation of applications, applications themselves are open to reverse engineering. In the following, we describe reverse engineering techniques for Android applications that analysts perform commonly.

3.2.1 Decompilation

Decompilation is the process of translating compiled code back into source code [190]. Although this translation rarely leads to a 1:1 representation of the original source code, the reverse engineered code is usually enough for an analyst to gain a comprehensive understanding of an application's logic and control flow [122] [190].

In the case of Android, decompilation aims to translate the bytecode stored inside DEX files contained in APKs into easily readable Java code, regardless whether the application was written in Java or Kotlin. Compared to machine code, Java and Dalvik bytecode, which is later interpreted by a virtual machine, contains much information of the original source code, such as variable and method names [189] [190]. In addition, applications are installed on the users' devices itself, allowing users to directly access and analyse them. As a result, decompilation is especially easily feasible on Android and requires special attention, as decompilation usually leads to an application's code being revealed, which could put intellectual property at risk.

Decompilation is performed due to various reasons. Analysts might be trying to extract confidential information from the application that is not intended to be revealed to the user, such as cryptographic keys [222]. Malicious reasons for decompilation include the intention to uncover implementation details, e.g. in order to pirate and redistribute a paid application [172]. Furthermore, analysing the implementation of an application could allow a malicious actor to create a malicious, repackaged version thereof.

In order to make application decompilation less effective, *obfuscation* can be employed.

3.2.2 Man-in-the-Middle Attack

A majority of Android applications perform network requests in order to communicate with remote servers. As a result, secure network communication should not be neglected.

One of the most common attacks on network traffic is the *man-in-the-middle (MITM)* attack [60]. This attack allows an attacker to intercept messages sent between two or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

more communicating parties [76]. In a passive attack, an attacker might only eavesdrop on the communication, while an active attack allows modifying and/or blocking network traffic [91] [256]. Besides the malicious background of MITM scenarios, malware analysts typically observe network traffic of applications as well. By doing so, important information about transmitted data and remote servers can be acquired.

MITM attacks are possible under various circumstances. For example, open access points or *evil twins* (fake access points that trick users to connect to it) are commonly used for MITM attacks [255].

While there are many different possible attack scenarios, according to Conti et al. [76], a common scenario involves intercepting the public keys of the victims when they are trying to initialize a secure connection and is illustrated in Figure 3.7. First, the attacker intercepts the two public keys of the victims (M1 and M2) and returns his/her public key to them (M3 and M4). As a result, the two communication parties still believe that they received each others' keys. Afterwards, one victim sends a message – encrypted with the attacker's public key – (M5) to the other user. However, the attacker is able to intercept and decrypt the message with his private key. Finally, the attacker can redirect – and optionally modify – the message to the intended recipient (M6).

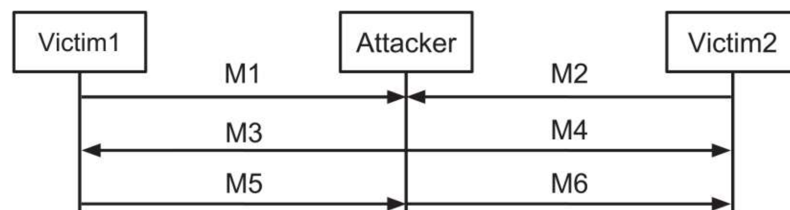


Figure 3.7: Man-in-the-middle attack scenario [76].

Nowadays, HTTP is usually replaced by HTTPS, i.e. HTTP over SSL or HTTP over TLS, the SSL protocol's successor [219], as it represents the de facto standard for establishing secure network connections. HTTPS prevents eavesdropping and was designed for bringing confidentiality, message integrity, as well as authentication to HTTP connections [71]. Although HTTPS minimizes the risk of MITM attacks, they cannot be prevented completely [58] [71].

A MITM attack on HTTPS could look like the following. Usually, SSL/TLS sessions start with the *SSL/TLS handshake* – a simple message exchange between client and server. The client initiates the exchange by sending a *client hello* and verifies the identity of the server [192]. This identity verification is mostly performed through digital signatures based on X.509 certificates that are signed by a certificate authority (CA) [60] [215]. After the client connects to the server, the server transmits the certificate to the client who then validates the certificate [91]. This validation process is critical and can lead to an insecure SSL/TLS connection in case it is flawed or manipulated. For example, a flawed validation process could involve trusting and accepting all certificates [192]. A

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

similar behaviour – namely passing the verification process with an invalid certificate – occurs when the victim ignores potential security warnings [76]. Furthermore, on most operating systems, including Android, all pre-installed CAs are trusted per default [30]. If any of these CAs issue a malicious certificate or a malicious certificate is installed explicitly, connections are at risk of man-in-the-middle (MITM) attacks [30].

In order to prevent MITM attacks to a certain degree, *TLS certificate pinning*, integrated in many anti-reversing tools along with other mechanisms and described in subsection 3.3.2, can be employed.

3.2.3 Rooting

Rooting describes the process of obtaining the highest user privileges ("root") in the operating system [188]. Therefore, rooting is not a reverse engineering technique as such, but often a prerequisite for further reverse engineering techniques described in this chapter.

In Android and typically in other Linux systems as well, "root" privileges are held by the *root* user [188]. To switch to the root user, the `su` binary can be used, which is installed during the rooting process [234]. When `su` is invoked, a root management application (such as *Magisk* [241] or *SuperSU* [235]) typically shows a confirmation dialogue before root privileges are granted to the process/application.

Rooting is generally categorised into two types [188] [263]:

- *Soft roots* aim to gain root privileges on an already booted system. They rely on exploiting vulnerabilities of the kernel or system processes running as root. However, soft roots are not persistent per se, as the integrity of the system is checked on each reboot.
- *Hard roots* are persistent but typically require the support of manufacturers for unlocking the bootloader. They are performed by directly flashing the `su` binary on the device or by replacing the operating system with a new one that has `su` installed [188] [176].

With root privileges, many actions that are normally not possible due to limited permissions, e.g. killing processes or accessing protected directories, can be performed [244] [62]. Therefore, rooting allows users to fully explore and customize their devices [213]. Furthermore, root access is often an requirement for reverse engineering tools (see section 3.4) to work properly [59].

However, rooting entails the risk of weakening the device's security, as it undermines Android's security model [123] [249]. On the one hand, malicious applications could simply request root access, instead of having to exploit a kernel or system vulnerability to gain root privileges [234]. Inexperienced users might grant the requested root permissions, allowing malware to gain control over the whole device [234]. On the other hand, malware

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

could exploit vulnerabilities within the `su` binary or root management application itself to gain root access [234].

Due to the security risks that are implied by a rooted device, developers of applications handling sensitive data often employ *root detection* (see subsection 3.3.3) to be able to react accordingly. In addition, root detection is also implemented by developers who strive for protecting their application logic, since reverse engineering tools often require root privileges.

3.2.4 Repackaging

Repackaging refers to the process of modifying an existing application and distributing it [177]. Typically, attackers download an existing (usually popular) application. Afterwards, the application can be reverse engineered, e.g. through decompilation. At this stage, nearly arbitrary modifications can be performed, including removing or redirecting application earnings and injecting malicious payloads or advertisements [66] [178]. Finally, the application is redistributed. Therefore, repackaging is a form of application cloning/plagiarism [143].

Repackaging is especially commonly employed within the Android ecosystem, as applications can easily be distributed on the official store by registering as an application developer and self-signing the application [135]. In addition, applications can also be distributed without relying on application stores through sideloading.

From a reverse engineer's perspective, unpacking and modifying an application allows disabling potential further anti-reverse engineering mechanisms.

In order to protect applications from being modified and/or repackaged, *tamper protection*, described in subsection 3.3.4, can be employed.

3.2.5 Hooking and Debugging

Hooking allows running arbitrary code within an application by hooking function calls and inserting additional functionality [124]. This way, the behaviour of applications can be changed completely. For example, method parameters, return values or whole implementations may be changed [227].

Although debugging is primarily used during development, it can also be used for altering the control flow and return values [124] [227]. On Android, two types of debuggers can be utilised: *Native code debuggers* or the Java-based debugger building upon the *Java Debug Wire Protocol (JDWP)* [193] [124]. The former typically makes use of `ptrace` [158] to directly attach to the process of the target application [65] [124]. In contrast, JDWP was designed for communication between the debugger and the JVM [65]. The ART/DVM also supports JDWP, which allows debugging Android applications using the same protocol as long as the `android:debuggable` attribute within the `AndroidManifest.xml` is set to `true` [65] [226].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Both techniques – hooking and debugging – are often used for analysing applications in-depth. The techniques can also be employed to circumvent security/anti-reversing mechanisms, such as root detection [231].

To prevent applications from being hooked or debugged, *hooking and debugging detection*, see subsection 3.3.5, can be employed. Besides developers, who use detection mechanisms to protect their applications, malware developers often employ hooking/debug detection to hinder researchers from analysing their malicious applications with the help of hooking or debugging tools [242].

3.2.6 Emulation

For easier analysis, applications are often ran in *emulators* instead of physical devices. This allows inspecting and resetting the application/system state or monitoring how the application operates [124]. In addition, emulators provide analysts with a controlled and isolated environment, which can be used for further dynamic (malware) analysis [202] [247].

As a result, malware developers tend to employ *emulation detection* (see subsection 3.3.6) to force researchers to avoid the usage of emulators for analysis purposes.

3.3 Android Application Anti-Reverse Engineering Mechanisms

In this section, we introduce anti-reverse engineering mechanisms often implemented by anti-reverse engineering tools aiming to prevent the previously described application reverse engineering techniques as far as possible.

3.3.1 Code Obfuscation

Code obfuscation is the process of changing the appearance of code into a form that is difficult to analyse. On Android, the compiled code is stored within one or more DEX files. Without obfuscation, it is trivial to analyse the human-readable Dalvik bytecode. With obfuscation applied, significantly more resources are required to gain a sufficient understanding of an application's logic as – depending on the level of obfuscation – the code becomes barely readable. As a result, code obfuscation helps protecting sensitive logic and intellectual property as well as preventing piracy [92]. In addition, obfuscation can hinder malware developers from publishing malicious versions of legitimate applications through repackaging (see subsection 3.2.4).

A common tool that employs basic obfuscation techniques is *ProGuard* [117], an open-source code transformer that compresses, optimizes and obfuscates bytecode [36]. It was integrated into the Android software development kit until Android switched to the *R8 compiler* to fulfil the tasks ProGuard was used for previously [36].

In the remainder of this section, we present three common obfuscation techniques.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Identifier Renaming

Usually, programmers choose meaningful names for identifiers (variables, method names, etc.) to achieve better readability. However, meaningful identifiers are also useful for reverse engineers to understand the functionality of an application more easily. As a result, obfuscators tend to rename identifiers of an application to meaningless names such as `ab`, `ba`, and alike [106]. Some obfuscators choose short strings in lexicographic order, others may generate strings in a completely random manner [182] [74]. Although renamed identifiers are usually easy to identify due to their length and meaninglessness, identifier renaming makes it harder to understand the semantics of an application. Listing 3.1 shows an example for identifier renaming, where non-descriptive names are assigned to two class variables.

```
1 public class ExampleClass {
2     String someString = "...";
3     int someInt = 1;
4 }
```

```
1 public class ExampleClass {
2     String aaa = "...";
3     int aab = 1;
4 }
```

Listing 3.1: Identifier renaming example.

Control Flow Obfuscation

Control flow obfuscation aims to hinder static analysis by changing the logical control flow of a program. This way, the execution paths of an application are changed while maintaining the intended functionality. Control flow obfuscation can be realised through (a combination of) several ways. For example, dead code that is never executed, e.g. additional methods or conditional statements that are never fulfilled, can be inserted [139]. Furthermore, loop conditions can be modified or extended by complex statements that have no influence on the result, i.e. the number of iterations remains the same.

Listings 3.2 and 3.3 illustrate a simple example for control flow obfuscation. The `for`-loop has been modified to run backwards and the reference to the variable `i` has been adjusted accordingly. In addition, an `if`-statement with a condition that is never fulfilled has been inserted.

Further control flow transformation techniques include the addition of redundant operations and reordering of statements [55] [54]. In addition to several other techniques, *control flow flattening* is an advanced technique that moves all function bodies, loops and conditional branches into a single loop that controls the program flow by using a switch or multiple if statements [141] [246].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Listing 3.2: Control flow obfuscation example: original code.

```
for (int i = 10; i > 0; i--) {
    if (i % 11 == 0) break;
    System.out.println(10 - i);
}
```

Listing 3.3: Control flow obfuscation example: obfuscated code.

Obfuscation through Reflection

Reflection is a Java programming technique that allows invoking methods dynamically. The goal of using the reflection mechanism for obfuscation purposes is to hide methods and fields that the code is calling [115]. This is realised by removing the direct references to methods and instead retrieving them dynamically via the reflection API [196] [115]. Therefore, obfuscation through reflection modifies the control flow of an application and can be seen as a variant of control flow obfuscation. However, the primary focus is to hide method and field names.

Although reflection can also be used for non-obfuscation purposes, this technique can provide a powerful way to impede reverse engineering, especially when used in combination with other obfuscation techniques [115]. An example for obfuscation through reflection is presented in Listings 3.4 and 3.5.

```
ExampleClass example = new ExampleClass();
example.exampleMethod();
```

Listing 3.4: Reflection example: original code.

```
Object c = Class.forName("com.example.ExampleClass")
                .getDeclaredConstructor().newInstance();
Method m = c.getClass().getMethod("exampleMethod");
m.invoke(c);
```

Listing 3.5: Reflection example: obfuscated code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

String Encryption

String encryption aims to replace string constants by encrypted representations based on cryptographic functions [82]. During runtime, before a string is used, the decryption routine is called. Therefore, when statically decompiling an application, original/plain strings are not visible to a reverse engineer.

Encryption and decryption routines may use a large variety of ciphers, such as simple XOR operations, AES, etc. [107]. In addition, to make implementations more robust, *white-box cryptography (WBC)* is often employed. WBC aims to protect cryptographic keys even if the attacker has full insights into the internal algorithm details [68].

Encrypting strings makes it more difficult for reverse engineers to understand program flow and logic, as (plain text) strings in source code are usually excellent reference points. Furthermore, string encryption helps hiding sensitive data, such as API endpoints, to a certain degree.

Listings 3.6 and 3.7 show a simple example of string encryption, with the former representing the original, unobfuscated code, and the latter showing its obfuscated counterpart. In this example, an additional class and method `StringEncryption.decrypt()` was added, which allows decrypting encrypted strings during runtime.

```
1 public class ExampleClass {
2     String url = "https://example.org";
3     String apiKey = "ABC1234567890";
4 }
```

Listing 3.6: String encryption example: original code.

```
1 public class ExampleClass {
2     String url = StringEncryption.decrypt("/&Wi(rsdhu");
3     String apiKey = StringEncryption.decrypt("(/&ajshd");
4 }
```

Listing 3.7: String encryption example: code with encrypted strings.

Class Encryption

Class encryption aims to encrypt code on a per class basis. A typical approach of class encryption mechanisms applied to Android applications is illustrated in Figure 3.8. Note that in this illustration, we assume that the APK only contains a single `classes.dex` file. The depicted approach encrypts the executable code stored inside the `classes.dex` file and stores the encrypted code as a resource/asset within the newly created protected APK [115]. The initial DEX file is replaced with an unpacking routine that is responsible

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for decrypting and loading the original code at runtime, usually immediately after the application was started [265] [170]. On Android, dynamic class loading can be implemented using the `DexClassLoader` API [22], which provides the functionality to load DEX files during runtime [129]. This process is similar to the functionality of *packers*, which were originally designed to compress software that unpacks itself during runtime [84].

Class encryption prevents usual decompilation approaches, as only the code for decrypting the original code is present in its original, unencrypted form. As with string encryption, class encryption implementations often make use of white-box cryptography to prevent reverse engineers from manually decrypting the encrypted classes.

However, class encryption is also often abused by malware, as signature-based detection can easily be evaded due to the only code being visible is the decryption routine [216].

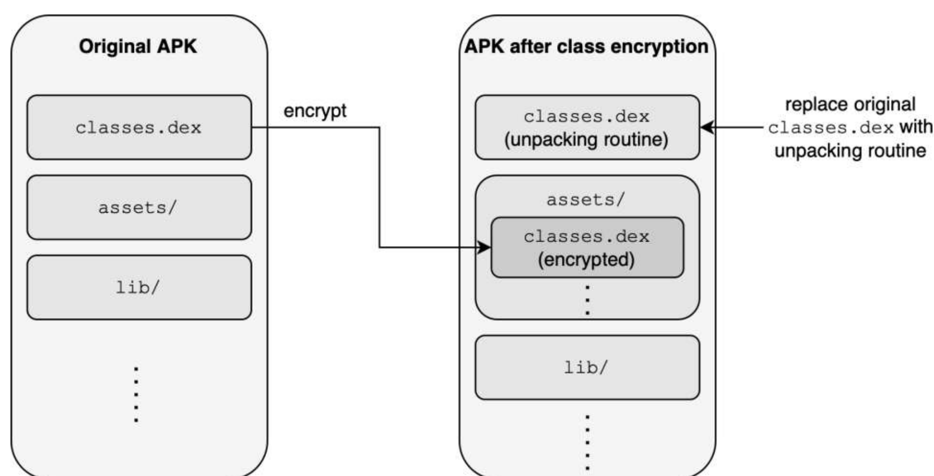


Figure 3.8: Typical class encryption process on Android.

3.3.2 TLS Certificate Pinning

TLS certificate pinning is a technique to mitigate MITM attacks over HTTPS. The main idea of TLS certificate pinning is to explicitly specify, i.e. "pin", certificates that should be trusted and refuse connections using other certificates [90]. Therefore, an application's secure connection is restricted to specific certificates. This specification of trusted certificates can be carried out in two ways. On the one hand, developers may directly specify the trusted TLS certificates [226]. However, when the specified certificate is replaced, e.g. due to expiration, an application update with the newly pinned certificates has to be distributed in order to maintain functionality. Therefore, developers also have the possibility to specify the hashes of the trusted certificates' public keys [226]. Thus, even when certificates are swapped out, the underlying public key usually remains static, making swapping out certificates much easier for developers [252].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A typical TLS certificate pinning implementation process is illustrated in Figure 3.9. First, when the client device initiates the communication with the server, the certificate is transferred to the client [91]. Afterwards, the client hashes the public key of the retrieved certificate. The hash is then compared with the one that is stored in the application [215]. If and only if the two hashes match, the communication is established. Therefore, even if the client tries to establish a connection using, e.g. a fraudulent certificate that was issued by a operating system-wide trusted CA, TLS certificate pinning prevents establishing the connection, as the hash comparison fails.

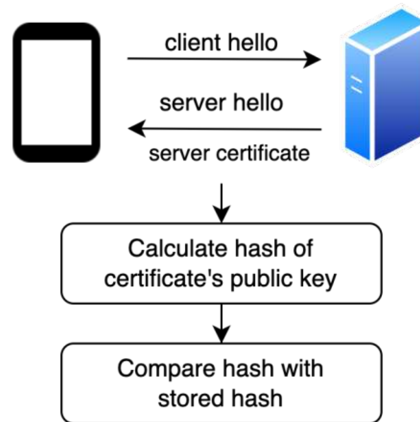


Figure 3.9: TLS certificate pinning process, adapted from [215].

On Android, trusted CAs are stored in the `TrustManager` [40] [192]. Since Android 7.0, the built-in pinning mechanism can be used by defining a *network security configuration* [30] and providing a set of public key hashes of certificates without having to write custom code [191]. An example configuration for pinning certificates is illustrated in Listing 3.8.

```xml
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">example.com</domain>
        <pin-set>
            <pin digest="SHA-256">7HIpactkIA...</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

Listing 3.8: TLS certificate pinning using Android's network security configuration [30].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3.3 Root Detection

To ensure that an application is running in a safe/unrooted environment, *root detection* can be employed. Typically, the corresponding code that performs various checks to determine whether a device is rooted is executed at the start of the application. In case the results imply a rooted device, an action – such as showing an appropriate error message and exiting the application – can be performed.

Root checks can be implemented using various methods. In the following, we will present the most common methods according to a study conducted by Sun et al. [234]. The presented methods are commonly used by root detection tools, such as the open-source library *RootBeer* [223].

Checking Installed Applications

Specific applications, such as root management applications like *Magisk* [241] or *SuperSU* [235], are commonly installed on rooted devices. On Android, the `PackageManager` class [31] allows to retrieve a list of the package names of all installed applications. Common root detection implementations iterate through this list and check whether certain (root-indicating) packages are installed.

Instead of using the `PackageManager` Java API, the command `pm list packages`, which lists all installed packages, can also be used.

Checking Existence of Files

To switch to the privileged "root" user, the `su` binary is used usually. As this binary is not present on unrooted devices, the existence of this file is a good indication of a rooted device.

Checking the existence of files can be done through various ways, such as using Java's `File.exists()` [25] method. As the `su` binary may be present in one of several directories, multiple directories, such as `/system/bin/su`, have to be checked.

Alternatively, `which su` can be executed which outputs the path to the superuser binary in case it exits. Otherwise, an error code is returned.

Checking System Properties

Android stores several different system properties, which can be accessed using the `getprop` command or the `System.getProperty()` method of the Java API. Some of the stored properties indicate a rooted device. In particular, the `ro.debuggable` and `ro.secure` properties set to `1` and `0`, respectively, indicate possible root privileges [188]. In addition, the `ro.build.tags` property might be equal to `test-keys` when using an inofficial Android build (in contrast to `release-keys` on official Android builds).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Checking Directory Permissions

Rooting a device might change some permissions of certain directories. For example, the `/system` directory is usually restricted to read-only access. If said directory turns out to be writeable, e.g. by using `File.canWrite()` [25], a rooted device can be concluded.

Checking Processes

Similar to checking installed applications, the currently running processes can also be checked. If one of the processes is likely to be a root management application, such as *Magisk* [241], a rooted device can be concluded.

For retrieving the running processes the `getRunningAppProcesses()` method of the `ActivityManager` class [6] can be used. In addition, executing the `ps` command returns a list of the running processes as well.

3.3.4 Tamper Protection

Tamper protection or *integrity protection* aims to ensure that an application has not been altered by a third party [227]. A basic approach for checking the integrity of an application consists of checking the signature of the APK [124]. If the signature does not match the expected one, the application was very likely modified. For this approach, it is necessary to embed the original signature/digest at build time, e.g. as a constant. Most implementations try to hide the check inside the application's code to prevent attackers from easily bypassing the detection.

More sophisticated approaches make use of feature extraction, filtering and similarity analysis [262]. In addition, checks are often embedded in multiple locations inside the application, thus making it harder for attackers to circumvent the detection of an altered application [218].

However, repackaging might also be utilised by security researchers to enable easier analysis of potential malicious applications. For example, anti-analysis techniques – such as root detection – could be disabled by modifying the corresponding code segments. As a result, malware authors often employ tamper protection as well.

3.3.5 Hooking and Debugging Detection

Hooking and debugging detection can be employed to prevent the execution of an application in case a hooking or debugging framework is attached.

Hooking tools can be detected by checking installed application packages, files or processes, similar to checking whether a device is rooted [198]. As there are different dynamic code instrumentation tools (see subsection 3.4.2) that provide hooking functionality available, further detection techniques depend on the specific tool that is used.

The detection of debuggers considers the two possible types of debugging on Android: Native code and Java-based debugging. One method for detecting native code debugging

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is measuring the execution time between two points in the control flow [99]. A large difference in time is an indication that the execution of the process was paused (e.g. through breakpoints) [99]. Another possible indicator is the `TracerPid`, which can be retrieved by inspecting the status file of the process (`proc/self/status` [157]) [198]. If a debugger is attached, the `TracerPid` has a value different from `0` [198].

A simple method to detect debugging using JDWP is checking whether the `debuggable` attribute within the Android manifest is set to `true` as this should never be the case with unmodified published applications [226]. In addition, the `isDebuggerConnected()` method of the `android.os.Debug` class [20] can be used to check whether a debugger is attached [198]. Furthermore, the technique of measuring the time difference between two points in the executed code that was pointed out previously can also be applied to detect JDWP based debugging [198].

In addition to detecting hooking or debugging tools, applications might aim to prevent the usage of such tools in advance. A common prevention technique leverages the fact that debuggers and hooking frameworks typically make use of `ptrace` [158] to observe and control a target process. However, only one tracer is allowed per process, meaning that if the process attaches to itself, no debugging/hooking tools that rely on the usage of `ptrace` can be attached [124].

3.3.6 Emulation Detection

Emulation detection aims to detect the execution of an application within an emulator. The detection of an emulated device is possible due to the fact that recreating a complete and real emulated system is difficult [227]. Thus, emulation detection techniques focus on detecting small differences of certain system artefacts and properties – statically and dynamically [133] [227].

Static checks inspect system properties, such as device ID, serial number, model, manufacturer, etc. [169] [198]. Dynamic checks might retrieve values from different device sensors, including the accelerometer, gyroscope, and GPS [202]. In general, static sensor values indicate emulated devices.

Although – like with other anti-reversing mechanisms – emulation detection can be bypassed, e.g. by simulating varying sensor values, it represents another necessary step and therefore impedes application analysis.

3.4 Reverse Engineering Tools

The following section provides an overview of a variety of tools that are typically used for reverse engineering applications and Android applications in particular.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.4.1 Decompilers and Disassemblers

Decompilers are tools that attempt to perform the inverse process of a compiler [69]. They take an executable (machine-dependent) program compiled to machine code as an input and aim to produce code written in a high-level language that behaves the same as the original executable given [69].

Disassemblers perform a subset of this reverse operation [250]. Instead of transforming a program into a high-level language, the program is transformed into a human readable representation of machine code, called *assembly* [250].

Decompilation in the case of Android aims to translate the compiled Dalvik bytecode (instead of machine code) stored in DEX files into easily readable Java source code. Analogically, Android disassemblers transform bytecode into a human readable representation called *smali* [103].

In order to perform decompilation and disassembly on the Android platform, a wide range of tools exists, with the most popular ones being introduced in the following:

apktool

apktool [243] is an open-source command-line tool for reverse engineering Android applications [83]. As some resources are encoded into binary XML format, simply unzipping APKs would result in some files being unreadable [198]. When using apktool to unpack APKs, XML files are decoded to text-based XML format, resources are extracted, and DEX files are disassembled to smali representations [198] [83]. The unpacked resources (e.g. layout XML files) and smali code can be modified and eventually be reassembled to produce an APK [83].

dex2jar

dex2jar [210] is an open-source project providing a set of tools for working with Android DEX and Java class files [83]. Typically, dex2jar is used to convert DEX files into the Java Archive (JAR) format. This conversion allows using common Java decompilers, such as *JD* [131], for decompiling Android applications.

Further features include reading and writing to DEX files as well as disassembling DEX files to their corresponding smali representation and vice versa [52].

Java Decompiler (JD) and jadx

The *Java decompiler project* [131] consists of a few tools aimed to decompile and analyse Java bytecode. There also exists a standalone version with a graphical user interface (JD-GUI) that allows browsing the decompiled source code of JAR files. In combination with *dex2jar*, JD is also suitable for decompiling Android applications.

In contrast, *jadx* [229] directly produces Java source code from DEX or APK files. In addition, binary XML files and other resources are decoded. Furthermore, the graphical

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

user interface provides additional functionality, such as jumping to declarations or searching for specific statements and strings.

IDA Pro and Ghidra

IDA Pro [126] is a commercial disassembler, which is commonly referred to as the industry standard reverse engineering tool [220]. In addition to the commercial license, the developing company Hex-Rays also offers a limited-functionality freeware version [85].

Ghidra [186] is an open-source reverse engineering framework developed by the National Security Agency (NSA) and was released to the public in 2019 [86] [220]. Ghidra encompasses a disassembler, decompiler as well as a scripting engine for advanced use cases [198].

Both – IDA Pro and Ghidra – are primarily known to be reverse engineering frameworks for disassembling programs compiled to machine code. For one, this allows disassembling and analysing native libraries of Android applications. Additionally, both tools are capable of opening, disassembling, and decompiling DEX files directly.

3.4.2 Dynamic Code Instrumentation Tools

Dynamic code instrumentation describes the process of modifying the instructions of a program during runtime [214]. Dynamic code instrumentation tools allow hooking function calls. Through hooking, a function call can be intercepted before the original function is called. Instead of the original function, custom code is executed, which may or may not call the original function implementation. Thus, hooking allows running arbitrary code within an application. Due to the possibility of changing the behaviour of applications completely, e.g. through substituting return values, dynamic code instrumentation is a powerful technique for analysing the functionality of applications as well as applied anti-reversing tools and their mechanisms.

Frida

Frida [95] is a popular, state-of-the-art dynamic code instrumentation tool, which allows the instrumentation of native code and Java bytecode. It is also the foundation of various other tools, such as the mobile exploration toolkit *objection* [224]. Objection offers some built-in functionality for interacting with mobile applications, e.g. bypassing TLS certificate pinning or dumping memory.

Frida consists of multiple components, such as [98]:

- *frida-core*: The *frida-core* component contains the main injection code.
- *Gum*: Frida uses the instrumentation library *Gum*, which includes a JavaScript engine.
- *frida-helper*: The *frida-helper* is responsible for the injection of the *frida-agent*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- *frida-agent*: Frida injects a shared library, called *frida-agent*, into the target process. This library receives commands and forwards them to the instrumentation library.

- *frida-server*: In order to make the *frida-core* accessible for remote hosts via TCP, the *frida-server* component is needed.

By injecting a JavaScript engine into the target process, Frida allows to write the instrumentation code in the high-level language JavaScript. For injecting the agent, Frida first leverages `ptrace` [158] to interact with the target process and store the current program status. Afterwards, Frida allocates memory for inserting the loader code and copies the loader to the corresponding memory location. Next, the loader is executed to inject an agent. Finally, the program execution continues normally by restoring the previously stored program status.

Further, Frida is organised as a client-server architecture, which allows instrumenting processes running on a remote machine. In the case of Android, the remote machine represents an Android device or emulator. Frida supports various operating systems, including Android, by providing the corresponding server binaries compiled for different architectures. The server binary requires root access on the device and allows instrumenting any installed application. If root access is not available, the *frida-gadget* library can be used. This shared library has to be embedded into the target application by unpacking, modifying and repacking the APK manually.

For hooking arbitrary functions, Frida offers the Interceptor API [96], which allows hooking functions by providing the memory address of the target function. Additionally, Frida allows hooking and modifying Java methods by specifying the exact class and method names. Furthermore, Frida's code tracing engine Stalker [97] allows tracing every executed instruction.

Xposed / LSPosed Framework

The *Xposed framework* [258] aims to change the behaviour of the system and applications without changing APKs by allowing developers to replace any method in any class of applications or the system [179]. The injection of custom behaviour is achieved by extending `/system/bin/app_process` – the binary which executes a runtime environment for Dalvik classes – by a JAR file [179]. Xposed itself only provides the functionality to modify methods. The actual functionality (i.e. modified behaviour) is implemented in *Xposed modules*, which can easily be published and shared with other users.

As Xposed is not actively maintained anymore, its successor *LSPosed* [104] has evolved. LSPosed provides similar functionality as Xposed and supports devices running Android 8.1 and above.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Cydia Substrate

Cydia Substrate [221] is a tool originally developed for customising iOS devices. Later on, a version for Android, supporting versions 2.3 through 4.3, was released. Cydia Substrate allowed modifying applications without requiring source code. Similar to the Xposed framework, Substrate only provides the toolkit for allowing instrumentation [179]. The modifications themselves are implemented in *substrate extensions*.

3.4.3 Diagnostic and Debugging Tools

Diagnostic and debugging tools allow monitoring and modifying the execution of processes. Simple monitoring might only consist of tracing executed system calls of an application. Debugging extends the idea of monitoring a process with the possibility of intervening in the execution of a program. For example, values of variables can be changed, which might cause a complete change of the program's behaviour. Thus, diagnostic and debugging tools are also crucial when analysing the behaviour of unknown software.

System Call Tracing With strace and jtrace

strace [160] is a Unix utility used to monitor system calls. One simple use case of `strace` is to specify a command that should be executed. During the execution, `strace` records the system calls that are called by the process.

Each output line contains the system call name, along with its arguments in parentheses and its return value after the equal sign. The following line is a snippet of the output when using `strace` on the command `cat /dev/null` as the manual page of `strace` [160] shows:

```
open("/dev/null", O_RDONLY) = 3
```

Tracing system calls can provide useful insights into the functionality of closed-source programs as many functions that are available in high-level languages rely on low-level system calls. For example, functions used for reading contents from files inevitably leverage `open` and `read` system calls. Therefore, we will perform system call tracing during the analysis of anti-reverse engineering tools in this work.

In addition to `strace`, *jtrace* [134], developed by Jonathan Levin, can be used when tracing system calls on Android. `jtrace` is comparable to `strace` enriched with some additional, Android-specific features. For example, `jtrace` is able to automatically parse *Binder* messages, thus providing insights into Android's inter-process communication. Furthermore, `jtrace` provides a plugin architecture, allowing to extend its functionality.

Debugging with Android Studio and GDB / LLDB

For debugging Android applications, Android recommends the debugging tools contained in *Android Studio* [9], which is the official IDE for Android application development [21].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In order to debug an application, the `debuggable` attribute in the Android manifest has to be set to `true`.

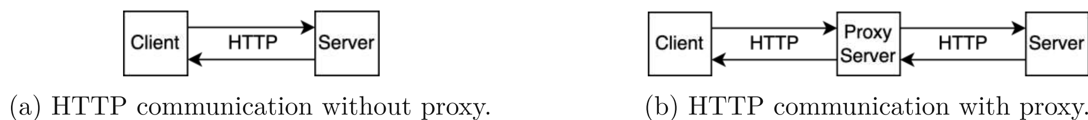
In addition, Android Studio also supports debugging native code as long as the source code is available. However, when confronted with unknown shared libraries that are part of Android applications, source code is usually not obtainable. In such cases, *GDB* [94] or *LLDB* [239] can be used.

GDB, the GNU project debugger, is a popular command line debugger for Unix systems. It provides various common debugging features, such as stopping on breakpoints, examining stack traces and modifying variables. For debugging applications running on remote machines, such as Android applications running on Android devices, `gdbserver` has to be started on the remote system [232]. This program allows connecting GDB running on another machine to a process via TCP.

LLDB is part of the LLVM project [240] and provides similar functionality as GDB although some commands differ. Using LLDB is Google's recommended way for debugging native code of Android applications [21]. In addition, the debugger integrated in Android studio is based on LLDB [21].

3.4.4 HTTP(S) Proxies

An HTTP(S) proxy server is an intermediary in the network between a client and server [236]. Thus, the client does not connect directly to the server, but instead sends its requests to the proxy server which *proxies*, i.e. redirects, the requests to the actual target server. This behaviour is illustrated in Figure 3.10.



(a) HTTP communication without proxy. (b) HTTP communication with proxy.

Figure 3.10: HTTP communication with and without proxy server.

Although there are various reasons for using proxy servers in the wild, e.g. to access external sources from within a firewall or to increase anonymity when accessing a target server, they can also be used for network analysis [168] [236].

As the whole network traffic is transmitted through the proxy server, it may act as a *man-in-the-middle*, thus allowing to intercept network requests in order to observe or even modify them. Besides the possibility of malicious usage, e.g. observing traffic of strangers, proxies are very commonly used when analysing applications.

However, the communication of most modern web and mobile applications is performed using HTTP over SSL/TLS (HTTPS) to prevent eavesdropping and tampering, e.g. as part of man-in-the-middle attacks. When using a proxy server, clients now directly connect to the proxy instead of the original server. As a result, the proxy server needs to create X.509 certificates through a custom certificate authority (CA). During the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

SSL/TLS handshake, the client tries to verify these newly generated certificates. However, the client usually rejects the connection as the custom CA is not trusted per default. Thus, in order to enable a successful verification, the custom CA certificate has to be installed and trusted on the client device. In the case of *TLS certificate pinning* being employed, further measures, e.g. bypassing the pinning mechanism, have to be taken in order to allow traffic analysis.

In the following, we present two popular tools that allow creating proxy servers for testing and analysing (Android) applications.

Burp Suite

Burp Suite [203] is a popular web application security testing tool developed by *PortSwigger* [204]. The suite provides a large feature set, such as automated security testing, fuzzing, and quality analysis of randomised values [212]. A fundamental feature is an HTTP(S) proxy server, which is not only suitable for web, but mobile applications as well. The proxy server, along with other helpful tools, is part of *Burp Suite Community Edition*, which is free to use. In addition to monitoring network traffic, traffic can also be modified, repeated, and analysed.

mitmproxy

mitmproxy [79] is an open-source HTTPS proxy. It is primarily developed as a command line tool providing the functionality to intercept, inspecting, modifying, and replaying web traffic. In addition, mitmproxy offers a graphical web interface and Python [211] API allowing to write add-ons and scripts. Furthermore, entire HTTP conversations can be saved for later replay and analysis.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Mobile Anti-Reverse Engineering Tools

This chapter gives an overview of the three anti-reverse engineering tools that are analysed in this work. While the following sections describe the tools in more detail, Table 4.1 provides a high-level comparison of the tools' offered mechanisms that were introduced in section 3.3.

Tool	Identifier Renaming	Control Flow Obfuscation	Obfuscation through Reflection	String Encryption	Class Encryption	TLS Certificate Pinning	Root Detection	Tamper Protection	Hooking and Debugging Detection	Emulation Detection
DexProtector [145]	•	•	•	✓	✓	✓	✓	✓	✓	✓
LIAPP [165]	•	•	•	✓	✓	•	✓	✓	✓	✓
DashO [205]	✓	✓	•	✓	•	•	✓	✓	✓	✓

Table 4.1: Feature comparison of DexProtector, LIAPP, and DashO.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1 DexProtector

DexProtector [145] is a mobile security solution for Android and iOS platforms developed by *Licel* [149].

According to the documentation [146], main features of DexProtector include:

- Encryption of several app components in order to impede static analysis and reverse engineering.
- Various environment checks to make dynamic analysis more cumbersome.
- Integrity checks to avoid code tampering and repackaging.

In addition to encryption of strings and classes, DexProtector also implements encryption routines for application resources. Furthermore, DexProtector offers a dedicated TLS certificate pinning mechanism, which makes it stand out against the other two analysed tools. However, even though the rough outline of the features [146] states that code obfuscation is applied, a closer look on the feature matrix [147] reveals that DexProtector only provides the possibility of renaming resource names as well as hiding JNI methods and refers to this process as "obfuscation". Nonetheless, the JVM-based codebase (i.e. Java or Kotlin) can be encrypted, provided that the tool was configured correspondingly. As a result, bypassing or reverting the class encryption mechanisms reveals unobfuscated, well-readable source code. Nevertheless, DexProtector can be used in combination with *R8* [36], in order to additionally perform basic renaming and obfuscation techniques.

DexProtector operates on APK, AAB (Android App Bundle), or AAR (Android Archive, used for libraries [19]) files as an input. In addition, DexProtector can be integrated into an existing build process using its plugin for *Gradle* [113], the standard build system used for Android projects. Based on the provided configuration in form of an XML file, the configured protection mechanisms are applied and a protected APK, AAB, or AAR file is generated as an output. DexProtector also supports signing the application as part of the protection process. Thus, the protected application is ready for distribution without the need for further processing.

4.2 LIAPP

LIAPP [165] is a cloud-based mobile protection tool for Android and iOS developed by *Lockin Company* [167].

Main features of LIAPP include [165]: Protection of code through class and string encryption, tampering protection, and several environment checks, such as debugging and rooting detection.

As LIAPP is cloud-based, no software has to be downloaded. Instead, the whole process of protecting an application is conducted using a web interface. After logging in, a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

new record for an application can be created where configuration options can be set. Afterwards the APK or AAB file that should be protected can be uploaded. Finally, after the protection process was completed, the protected application can be downloaded. However, in order to be able to distribute and install the application on devices, it has to be signed afterwards.

4.3 DashO

DashO [205], developed by *PreEmptive* [208], is an obfuscation and hardening tool for Java, Kotlin, and Android applications. Although DashO does not support class encryption, it supports identifier renaming and control flow obfuscation. In addition to the alteration of class, method, and variable names, DashO's identifier renaming implementation applies an additional technique, which PreEmptive refers to as *overload induction* [206]. Overload induction aims to rename as many methods as possible to the same name. As the name implies, this is possible due to *overloading*, i.e. creating multiple functions of the same name but different parameters. However, when using DashO for Android applications, DashO's renaming implementation is not available, as it expects renaming to be handled by *R8* [36].

Furthermore, DashO supports string and resource encryption as well as tampering protection. For Android applications, DashO implements several environment checks, such as root detection. Additionally, the tool supports watermarking in order to track unauthorized copies of software.

DashO offers a "wizard" with a graphical user interface for automatically integrating DashO into an existing Android project using a Gradle plugin [207]. Nevertheless, the wizard also allows to select an APK file that should be protected, without having to integrate DashO into the build process [207]. PreEmptive recommends to configure the protection mechanisms using the DashO graphical user interface, which generates an XML configuration file with the specified options [207]. Nonetheless, DashO can also be configured by modifying the XML file manually.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 5

Analysis of Mobile Anti-Reverse Engineering Mechanisms

This chapter analyses several anti-reverse engineering mechanisms provided by the anti-reverse engineering tools *DexProtector* [145], *LIAPP* [165], and *DashO* [205]. First, we present the general analysis approach and setup along with the tools we are using throughout the process. Afterwards, we thoroughly describe the performed techniques and results of the analysis. Finally, this chapter concludes with an overview of the differences between the analysed tools.

5.1 Analysis Approach and Setup

This section describes this work's analysis approach and setup.

5.1.1 Approach

This work follows the analysis approach illustrated in Figure 5.1. Step (1) consists of implementing an evaluation application, as described in subsection 5.1.2. Step (2) applies one of the three anti-reversing tools (DexProtector, LIAPP, and DashO) with one specific protection mechanism to the application.

Afterwards, step (3) analyses the protected evaluation application. As a first analysis step, *static analysis* is performed in step (3a). By using several decompilation and disassembling tools, such as *apktool* [243] and *jadx* [229], some insights regarding the implementation of the applied protection mechanism are gained. Afterwards, step (3b) employs *dynamic analysis* based on the results of (3a). The focus of (3b) rests on applying dynamic code instrumentation and function hooking using *Frida* [95], allowing to trace function calls and replace function implementations. Leveraging the analysis results of (3b), further static analysis (step (3a)) helps to gain further comprehension of the mechanism's internal

functionality. After the analysis of one tool with one specific protection mechanism, steps (2) and (3) are repeated with another tool and/or mechanism.

Subsequently, after we analysed all of the tools and mechanisms in question, step (4) works out the main implementation differences of the tools.

Based on the analysis results, chapter 6 describes step (5) that implements several bypassing approaches for each of the analysed mechanisms as well as functionality to automatically determine which of the anti-reversing tools this work focuses on was applied to a given application. Step (6), which is covered in chapter 7, conceptualises possible approaches for improving the analysed mechanisms and tools based on the previous analysis results and developed bypassing strategies.

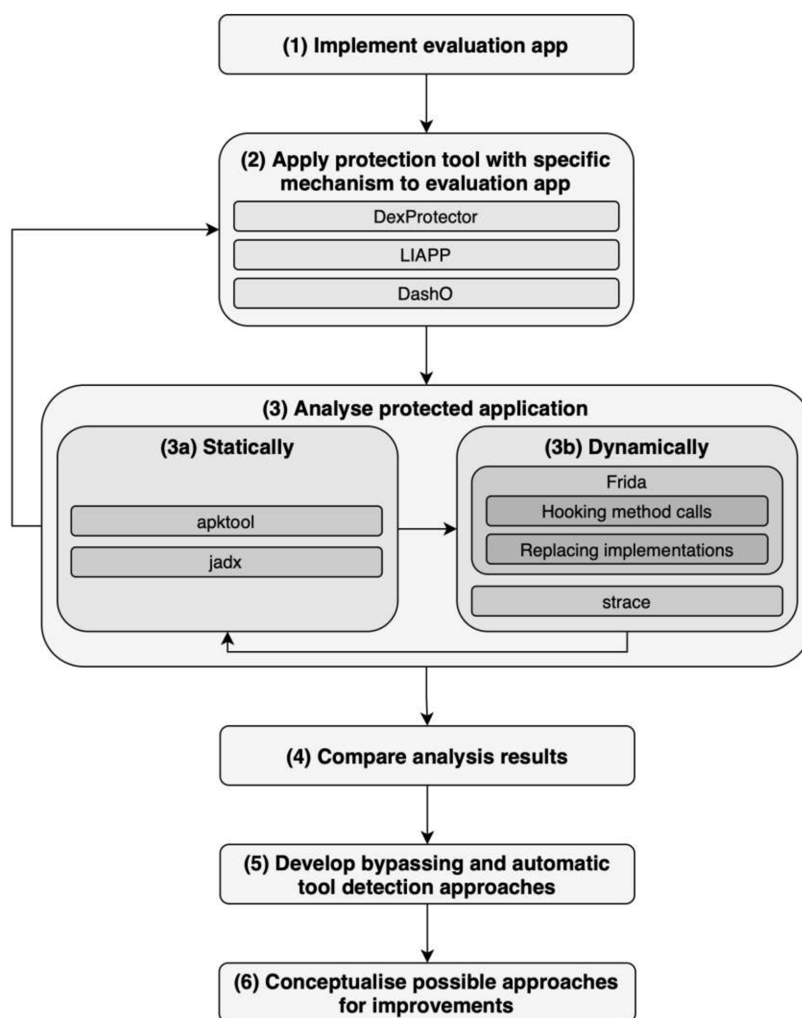


Figure 5.1: Analysis process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.1.2 Evaluation Android Application

In order to properly analyse the stated tools, a simple Android application with one activity, one fragment, as well as some basic features is implemented as part of analysis step (1) (see Figure 5.1). The application's main (and only) activity displays the result of the root check mechanisms offered by the tools. In addition, the application provides two buttons that can be pressed to perform an HTTPS `GET` or `POST` – depending on the button – request to the fake/mock REST API server *JSONPlaceholder* [245] using the networking library *Fuel* [137]. More specifically, `GET` requests are performed to `https://jsonplaceholder.typicode.com/posts/1` and `POST` requests to `https://jsonplaceholder.typicode.com/posts/`. The UI additionally contains two text fields where the id and post title, used for the `GET` and `POST` requests, respectively, can be specified. A screenshot of the described application is shown in Figure 5.2.

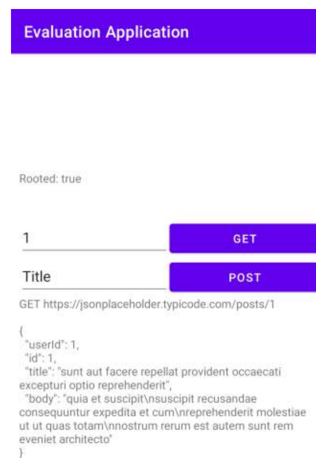


Figure 5.2: Screenshot of evaluation application.

5.1.3 Analysis Environment

For developing and building the Android application for analysing the applied anti-reversing tools, a MacBook Pro 2020 running macOS 12.6 was used. In addition, all reverse engineering tools that were used for this analysis, were installed and executed on this machine.

For executing different builds of the evaluation Android application, a Google Pixel 4a with Android 11 was used. The Android device was rooted using *Magisk* [241].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.2 Analysis Procedure and Results

This section describes the analysis procedure and results for each of the three anti-reversing tools.

5.2.1 DexProtector

For analysing DexProtector we use version 12.0.1.

The mechanisms that should be applied are configured in an XML file [144], as shown in Listing 5.1. In the shown example, only the string and class encryption mechanisms are enabled.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<dexprotector>
    <verbose>true</verbose>
    <optimize>false</optimize>
    <signMode>release</signMode>
    <licenseFile>dexprotector.licel</licenseFile>

    <stringEncryption/>
    <classEncryption/>
</dexprotector>
```

Listing 5.1: DexProtector: Example configuration file.

General

For gaining a rough overview of the internals of DexProtector, we first apply the tool with an empty configuration, i.e. no protection mechanisms are applied but DexProtector still processes the Android application.

As a starting point, we unpack the resulting APK with *apktool*. Navigating through the unpacked directories reveals that an `assets` directory containing four encrypted shared libraries – one for each architecture (`arm`, `arm64`, `x86`, and `x86_64`) – was added. Usually, shared libraries are stored inside the `lib` directory located in the root directory of an APK. However, DexProtector seems to decrypt and extract the needed library from the `assets` directory just before loading the shared library using `System.loadLibrary()` or `System.load()` calls [38]). This behaviour can be confirmed by decompiling the APK using *jadx*. In the application's main package, DexProtector added three additional classes. First, a custom exception called `MessageGuardException`, which is thrown on a few occasions, e.g. when the initialisation of the application failed or certain anti-reversing mechanisms take effect. Second, DexProtector added a custom `AppComponentFactory` [11] called `AppComponentFactoryDP`, which is used to manage the instantiation of elements defined in the Android manifest. Finally, DexProtector

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

added an `Application` class, which is instantiated before any other class upon starting an application [12]. The inserted `Application` class leverages several obfuscation techniques, such as identifier renaming and control flow obfuscation. Nonetheless, the method shown in Listing 5.2 can be identified. Note that we renamed the identifiers for easier readability. It includes a `System.load()` call (line 6) and a subsequent `delete()` call (line 11) on the file that was previously passed to `System.load()`. Furthermore, it is called with a string as parameter that is generated based on the contents of a file. Due to string encryption being applied, static analysis does not reveal the exact name of the file. However, this is a strong indication that the encrypted shared library inside the `assets` directory is read.

```
1  private void extractAndLoadLib(String strFromFile) {
2     File decryptedLib = initFile();
3     try {
4         try {
5             decryptLibToFile(decryptedLib, strFromFile);
6             System.load(decryptedLib.getAbsolutePath());
7         } catch (Exception e) {
8             throw new RuntimeException(valueOf("<encrypted string>")
   ↪  + decryptedLib + valueOf("<encrypted string>"), e);
9         }
10     } finally {
11        decryptedLib.delete();
12     }
13 }
```

Listing 5.2: DexProtector: Loading and deletion of shared library.

In order to further validate our assumptions, we use `strace` [160] for tracing system calls of the evaluation application. One possibility for doing so is to attach `strace` with the option to follow child processes enabled to *zygote*, the process that spawns other application/ART instances. Upon launching the application, the `strace` output shown in Listing 5.3 can be observed: First, the encrypted `so.dat` file located in the `assets` directory is read (line 2). Afterwards, the target file with the name `libdexprotector.<process ID>.so` is opened and the resulting file descriptor is passed to a subsequent `write` [162] call (lines 5 and 6). Finally, after the library was decrypted and written to the target `.so` file (lines 9 and 10), `unlinkat` [161] is called in order to remove the decrypted library again (line 15).

For retrieving the decrypted shared library, we create a *Frida* script that hooks the `System.load()` method. Whenever this method is called, a "sleep" instruction is inserted, allowing to copy the shared library to another location before it is deleted. This approach along with the decryption procedure is illustrated in Figure 5.3, where the injected operations are depicted in blue. Afterwards, *Ghidra* [186] can be used to inspect

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  # Reading encrypted library (com.risingapp.arm-v8.so.dat)
2  20336(evalapp): pread64 (45 </data/app/~~yP-Nn6KoLdtrug0-zbQV4A
   ↪ ==/a.b.app-ira19lTlpmzcGwt6QvatGw==/base.apk>,
   ↪ "assets/com.risingapp.arm-v8.so.dat",34,239516) = 34
3
4  # Opening target file (libdexprotector.20336.so)
5  20336(evalapp): openat  (AT_FDCWD,
   ↪ "/data/user/0/a.b.app/app_outdex/libdexprotector.20336.so",577)
   ↪ = descriptor 47
6  20336(evalapp): fstat64  (47
   ↪  </data/user/0/a.b.app/app_outdex/libdexprotector.20336.so>,{
   ↪  device: 1,2 inode: 19180 mode: 0100600 links: 1, uid: 10283 gid:
   ↪  10283 size: 0}) = 0
7
8  # Writing decrypted library
9  20336(evalapp): write (47
   ↪  </data/user/0/a.b.app/app_outdex/libdexprotector.20336.so>,
   ↪ "\x7fELF\x02\x01\x01\x00DPLF\x00\x00\x00\x00\x03\x00\xb7...",512
   ↪ ) = 512
10 20336(evalapp): write (47
   ↪  </data/user/0/a.b.app/app_outdex/libdexprotector.20336.so>, ...
11
12 # Several write and other calls for loading DexProtector library...
13
14 # Call unlink to remove loaded library
15 unlinkat  (AT_FDCWD,
   ↪ "/data/user/0/a.b.app/app_outdex/libdexprotector.20336.so",0) =
   ↪ 0
```

Listing 5.3: DexProtector: `strace` output when loading native library.

the library, revealing that it is heavily obfuscated and employing techniques similar to *packers*, which dynamically extract and load code during runtime [84]. Another notable aspect is that – for obfuscation purposes – the shared library does not explicitly declare the functions called by the Java side of the application via JNI [194]. Instead, the JNI function `RegisterNatives` [195] is used to register native functions during runtime. Nonetheless, leveraging Frida, the `RegisterNatives` function can be hooked in order to log the signatures of the registered functions.

Finally, the added `Application` class contains a method that is responsible for checking the signature of the application as shown in Listing 5.4. The `returns` method (lines 2 and 6) takes a non-readable/encrypted string as a parameter and presumably returns the original, readable string. The shown method (`azive`) retrieves the signature of the current application using `PackageManager` [31] (line 3), calculates the SHA-256 hash of the signature (line 5) and compares it with a predefined byte sequence (line 7). This byte

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

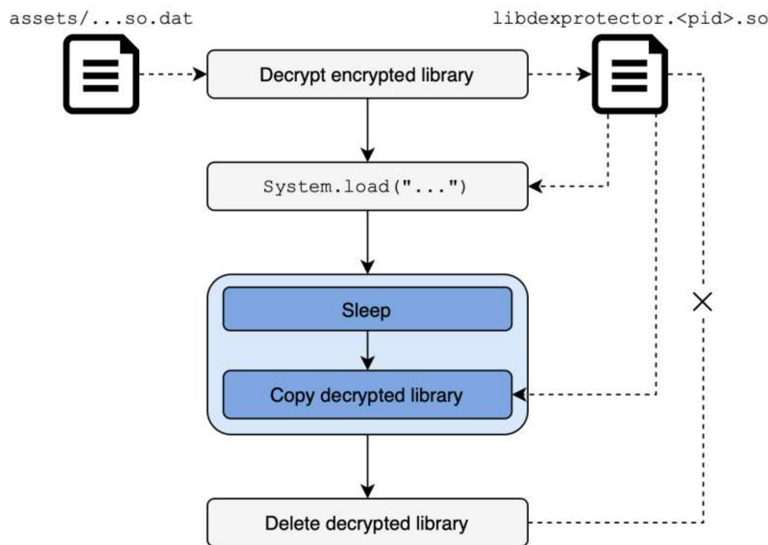


Figure 5.3: DexProtector: Extraction of native library.

sequence is equal to the hash of the certificate that was used for signing the application. If the hashes do not match, an exception is thrown and the application crashes (line 8). Thus, repackaging/resigning is prevented. Note that this check is always employed when building an application with DexProtector and there is no configuration option for this mechanism.

```
1  private void azive() {
2      MessageDigest instance = MessageDigest.getInstance(returns("..."
   ↪ )); // SHA-256
3      Signature[] signatureArr = getPackageManager().getPackageInfo(
   ↪ getPackageName(), 64).signatures;
4      // ...
5      String bigInteger = new BigInteger(1, instance.digest(
   ↪ signatureArr[0].toByteArray())).toString(16);
6      String returns = returns("...");
7      if (!returns.equals(bigInteger)) {
8          throw new RuntimeException(...);
9      }
10 }
```

Listing 5.4: DexProtector: Integrity check.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

String Encryption

For evaluating the string encryption feature of DexProtector only `<stringEncryption/>` was enabled inside DexProtector's configuration XML file.

Decompiling the protected application using *jadx* shows that the added `Application` class contains the additional method `public static native String s(String)` used for mapping encrypted to decrypted (plain) strings. DexProtector realises this method as a native call using JNI. As a result, the source code of the `s` method is not visible in the decompiled Java code but is instead part of the shared library that is decrypted and loaded during runtime. Furthermore, the `assets` directory also contains the file `se.dat` (possible abbreviation for "String Encryption") that might act as a key for the encryption process. Using `strace`, we can confirm that this file is mapped into memory during runtime.

By hooking the native `s` function with *Frida*, the calls and corresponding return values of the function can be logged. Analysing this output shows that the `s` function was called with strings containing various characters of the unicode table, not restricted to printable ones. While the function calls were already partially revealed by previous decompilation of the application, hooking also reveals the corresponding return values. For example, spotted parameters of the `s` function include `"\u00e3\u0090\u00b6"` and `"\u00e3\u0090\u00b7"`. The return values of the `s` function, i.e. decrypted/original strings, called with these parameters are `"MainActivity"` and `"https://jsonplaceholder.typicode.com/posts/"`, respectively. Thus, as subsequent byte sequences used as inputs lead to completely different outputs, it can be concluded that the string encryption mechanism is realised through a mapping of byte sequences to the corresponding original strings.

By applying the string encryption mechanism to only a handful of classes (using `<filters>` in the DexProtector configuration), it can be observed that the "encrypted" strings (i.e. the parameters passed to `Application.s()`) are generated deterministically. The classes are alphabetically traversed and each string occurrence is replaced by `Application.s()` calls with unicode characters as parameter that are increased by 1 for each string, i.e. `\u0000`, `\u0001`, `\u0002`, etc. Note that – due to this implementation – identical strings are assigned different "encrypted" values.

Class Encryption

For evaluating the class encryption feature of DexProtector only `<classEncryption/>` was enabled inside the configuration XML file.

Decompiling the protected application using *jadx* shows that the `Application` class was added again. Original source files, such as the `MainActivity` class, were removed. Instead, two `.dex.dat` files were added to the `assets` directory. Those files presumably correspond to the encrypted `classes.dex` file.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Lim and Yi [150], who analysed DexProtector's class encryption mechanism in 2016, describe the process as follows: First, the original (unencrypted) `classes.dex` file is moved to the `assets` directory and another `classes.dex`, which generates an "decryptor" APK, is added to the root directory (default location) of the APK. Because the primary `classes.dex` file is now the one that was newly added, the decryptor APK is generated and ran on the application's startup. Thus, on the application's startup, the encrypted `classes.dex` file located in the `assets` directory is decrypted and the original, unencrypted `classes.dex` file is restored. Finally, the original application is executed by loading the restored `classes.dex` file with `DexClassLoader` [22], for example. After loading the `.dex` file, it is removed again to make its extraction more cumbersome.

However, through hooking class loader functions and tracing system calls, we detected that DexProtector switched to a different approach over the years. Instead of writing the decrypted `classes.dex` to a file, everything happens in-memory directly. More precisely, function hooking and system call tracing reveals that the `DexFileLoader::OpenCommon` [4] function, which is invoked by `InMemoryDexClassLoader` [27], is used to directly open a DEX file from a given memory address. At the given address, memory was allocated and the decrypted content of the DEX file was copied to previously. Furthermore, decompilation shows that the protected APK does not contain any code for generating a decryptor APK or similar functionality. The only remaining class with actual functionality is the added `Application` class, which is mainly responsible for decrypting and loading DexProtector's shared library, leading to the conclusion that the described decryption procedure is performed by the loaded shared library.

TLS Certificate Pinning

For configuring the TLS certificate pinning mechanism of DexProtector the same structure as Android's built in network security configuration [30] as shown in Listing 3.8 is used. The `<network-security-config>` tag has to placed inside the `<publicKeyPinning>` tag of DexProtector's XML configuration file.

For analysing the TLS certificate pinning mechanism a custom CA certificate of *mitmproxy* [79] was installed on the Android device. Further, the IP address of a machine running mitmproxy was configured as a proxy server on the Android device. This setup allows intercepting HTTPS requests as long as no TLS certificate pinning is in place.

The user manual of DexProtector [148] states that the Android Security API – including `TrustManager` [40], which is used to enforce Android's built-in TLS certificate pinning through the network security configuration – is not used in DexProtector's TLS certificate pinning mechanism. Further, the manual states that, as a result, DexProtector is resistive against attacks on the `TrustManager`.

We verify this statement by overwriting the implementation of the `checkTrustedRecursive` method contained in the `TrustManagerImpl` class [102]. Usually, this

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

method recursively builds certificate chains while verifying each certificate. We replace the implementation with a single statement that returns an empty list of certificates, thus skipping the entire certificate verification process. This approach is commonly used for bypassing Android's built-in TLS certificate pinning mechanism, e.g. by *objection* [224], a tool built upon Frida. This approach was applied to the protected evaluation application. However, neither a simple Frida script replacing the `checkTrustedRecursive` method nor the TLS certificate pinning bypass functionality of objection allowed intercepting the requests with mitmproxy due to the `checkTrustedRecursive` method not being called. Instead, when a network request is performed, a stack trace, describing that a `CertificateException` [16] has been thrown due to a failed pin verification, was logged. The stack trace further points out that the exception is thrown inside of classes of *Fuel* [137], the networking library our evaluation application uses to perform the network requests.

In order to identify the specific method/class that is causing the exception to be thrown, we unpack both – the original, unprotected evaluation application and the one with the TLS certificate pinning mechanism being employed – using *apktool*. Afterwards, we calculate the differences of the unpacked applications. Besides the addition of DexProtector classes, one difference stands out. As shown in Listing 5.5 (line 7), within one of Fuel's classes, a `url.openConnection()` call, which is used for setting up the actual network connection [41], is replaced by a DexProtector specific method with a randomly generated name.

```
1  private final HttpURLConnection establishConnection(Request request)
   ↪  {
2      URLConnection uRLConnection;
3      URL url = request.getUrl();
4      Proxy proxy = this.proxy;
5      if (proxy == null || (uRLConnection = url.openConnection(proxy))
   ↪   == null) {
6          // originally 'url.openConnection()'
7          uRLConnection = Application.BuildConfig.ayzkAHq(url);
8      }
9      if (uRLConnection != null) {
10         return (HttpURLConnection) uRLConnection;
11     }
12     throw new NullPointerException(
   ↪   "null cannot be cast to non-null type " +
13         "java.net.HttpURLConnection");
14 }
```

Listing 5.5: DexProtector: Replacement of `url.openConnection()` call to enforce TLS certificate pinning.

As the original `url.openConnection()` call was not part of the standard Android

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Java/Kotlin API but a third-party library, we conclude that DexProtector actively scans for `url.openConnection()` calls in order to inject its custom certificate verification logic.

Further investigation reveals that the `Application.BuildConfig` class, which is accessed in line 7 of Listing 5.5, is not present when unpacking/decompiling the protected APK. Instead, the class is loaded at runtime as soon as it is needed, i.e. when an HTTP request is performed for the first time. This behaviour can be observed when enumerating all loaded classes and hooking the `DefineClass` method using a Frida script. The script prints the loaded classes at startup, which is only the `AppComponentFactoryDP` class, and hooks all of its methods, revealing that the `instantiateApplication` and `instantiateActivity` methods – used for instantiating the corresponding elements the names imply – are called. Furthermore, the JNI `DefineClass` method [195], which is used for loading a class from a buffer, is hooked. This procedure reveals that the `Application$BuildConfig` class is loaded when performing the first HTTP request. In addition, we can observe that several classes inside a dedicated `com.licel.dex-protector` package are loaded. These classes are used to perform the certificate verification process and eventually also the initial `url.openConnection()` call, provided that the verification was successful.

Due to `Application$BuildConfig` and other corresponding classes not being present when simply decompiling the APK, the classes must be extracted at runtime. In order to retrieve the classes, the approach that is also used for extracting the decrypted classes when using class encryption (see subsection 6.1.2) can be used. Note that – because the classes are only generated when the first HTTP request was sent – a request has to be sent before extracting the classes in form of DEX files. One of the extracted DEX files contains the generated classes that are specific to TLS certificate pinning. As an example, Listing 5.6 shows the `Application$BuildConfig` class, which contains the method that DexProtector calls instead of `url.openConnection()`. The method of this class checks whether protocol of the URL is equal to `"https"` (line 6). If this is the case, further action, i.e. verifying the certificate, is taken. The implementation of the validation is spread across several obfuscated classes, starting with `Application$Application`.

```
1  public class Application$BuildConfig {
2      public static URLConnection ayzkAHq(URL url) {
3          if ("http".equals(url.getProtocol())) {
4              return new Application$R$integer(url);
5          }
6          return "https".equals(url.getProtocol()) ? new
   ↪  Application$Application(url) : url.openConnection();
7      }
8  }
```

Listing 5.6: DexProtector: Generated class to enforce TLS certificate pinning.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Root Detection

For the root detection mechanism, DexProtector requires specifying three methods within the `<root>` element of the configuration file [148]:

- `negativeCheckCallback()` is called when the device has *not* been classified as rooted.
- `positiveCheckCallback()` is called when a rooted device has been classified as rooted.
- `doProbe()` collects the environment data and must be called before being able to use the check results. According to DexProtector's documentation [148], a call to a function that is responsible for detecting root is inserted at the `doProbe()` method at build time.

These methods are defined by specifying the class and method names of existing Java/Kotlin methods that should be used for the purposes described above. In addition, the two callback methods should have an `Object` parameter in order for DexProtector being able to pass additional information regarding the result of the root check.

Decompiling the protected APK using *jadx* reveals that a call to an `r()` method of a class with a randomly generated name was inserted at the beginning of the specified `doProbe()` method. Inspecting the class of the native `r()` method reveals several other, additional native methods, such as `d()` and `e()`. Some of them are used for other environment checks, such as hooking and emulator detection. In this work, we focus on the `r()` method, which is used for the root detection mechanism. It is called at the beginning of the specified `doProbe()` function, performs the root checks inside DexProtector's native library, and calls one of the two callbacks depending on the check's result, as depicted in Figure 5.4.

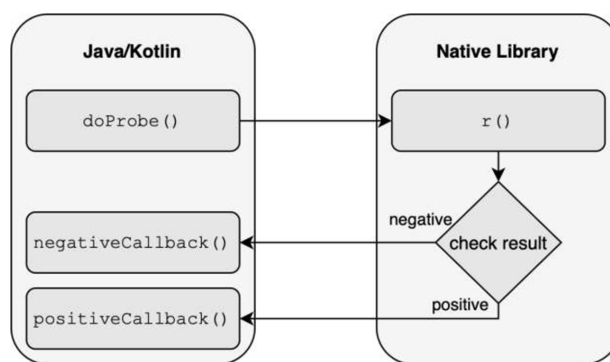


Figure 5.4: DexProtector: Root detection process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

By hooking the callbacks and logging the `Object` parameters with *Frida*, we can observe that the callbacks receive a JSON string (shown in Listing 5.7) that reveals a bit of information about the checks. On the analysis device three checks were positive (lines 6, 9, and 11). In the following, we analyse the internal implementations (within the native DexProtector library) of each of the properties/checks outlined in the JSON. However, due to the native library being heavily obfuscated, analysis thereof is not possible in an efficient way. Instead, we primarily trace system calls using *strace* for investigating the performed actions as root checks heavily rely on executing system calls [234].


```
1  {
2      "DetectTestKeys": 0,
3      "DetectRootManagmentApps": 0,
4      "DetectPotentiallyDangerousApps": 0,
5      "DetectRootCloakingApps": 0,
6      "CheckForSuBinary": 1,
7      "CheckForDangerousProps": 0,
8      "CheckForRWPaths": 0,
9      "CheckForMagiskFiles": 1,
10     "CheckForMagiskManagerApp": 0,
11     "CheckSuExists": 1
12 }
```


Listing 5.7: DexProtector: Root detection result in JSON.

Upon launching the application, we can observe that `stat` [159] is called several times to check if files indicating a rooted device are present. This primarily concerns the superuser binary (`su`), which can be located in various locations. Parts of the `strace` output is shown in Listing 5.8. On the analysis device the `/system/bin/su` binary was present, causing this particular `stat` call to succeed (line 4). When renaming this binary, the `stat` call now returns `-1`. In addition, renaming the binary causes the `CheckForSuBinary` and `CheckSuExists` properties of the JSON string to be set to `0`.

Further, the `strace` output reveals that DexProtector opens and reads `/proc/self/mounts`. On our analysis device the rooting software *Magisk* is installed and mounted into the application's process. Thus, the `/proc/self/mounts` file contains the string `"magisk"`. As a result, `CheckForMagiskFiles` is set to `1`, while the similar named property `CheckForMagiskManager` is set to `0`, which is assumed to have been used for indicating the presence of *Magisk Manager*, the predecessor of Magisk [241]. However, *SELinux* often blocks access to system paths, such as `/proc/self/mounts`. In order to bypass this restriction, DexProtector creates a service with the `isolatedProcess` property set to `true`. According to the documentation regarding SELinux policies [49] and the defined rules for isolated processes [110], services declared as `isolatedProcess` bypass certain SELinux restrictions, even when using SELinux' "enforcing" mode. In addition,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  ...
2
3  17970 10:53:39.398542 newfstatat(AT_FDCWD, "/system_ext/bin/su",
   ↪  0x7fc794c1f0, 0) = -1 ENOENT (No such file or directory) <
   ↪  0.000023>
4  17970 10:53:39.398606 newfstatat(AT_FDCWD, "/system/bin/su", {
   ↪  st_mode=S_IFREG|0755, st_size=170176, ...}, 0) = 0 <0.000054>
5  17970 10:53:39.398707 newfstatat(AT_FDCWD, "/system/xbin/su",
   ↪  0x7fc794c1f0, 0) = -1 ENOENT (No such file or directory) <
   ↪  0.000021>
6
7  ...
```

Listing 5.8: DexProtector: `strace` output when checking root-indicating files.

spawning an isolated process to perform the checks might bypass root hiding mechanisms, due to isolated processes might not being detected by root hiding tools [80] [108]. This service can be seen in the decompiled APK, provided that the root detection mechanism is enabled. In addition, the service is declared in the Android manifest.

When executing the application on our evaluation Android device, the properties `DetectRootManagmentApps` [sic], `DetectRootCloakingApps`, and `DetectPotentiallyDangerousApps` are set to `0`. Although the property names already provide some guidance regarding the functionality of the checks, assumptions can be confirmed using *jtrace* [134]. The `jtrace` output reveals that DexProtector uses Android's `PackageManager` [31] to check if any root management, hiding, or other "dangerous" applications requiring root are installed on the device.

The `strace` output further shows that system properties are accessed. System properties on Android provide information about the software and device, such as the product name and brand [111]. In particular, `strace` shows that the `ro.build.tags` property is accessed. This property is often equal to `test-keys` when using an emulator (i.e. developer build) or an inofficial build of the Android OS, as opposed to `release-keys` on Android builds provided by OEMs [234]. Further, DexProtector accesses the `ro.debuggable` property, where a value equal to `1` indicates possible root privileges [188]. We confirm this behaviour by installing the evaluation application on an Android emulator running Android 11. Inspecting the emulator's system properties reveals that `ro.build.tags` and `ro.debuggable` are set to `test-keys` and `1`, respectively. Launching the application and inspecting the returned JSON string shows that `DetectTestKeys` and `CheckForDangerousProps` are now both set to `1`, confirming our assumption.

Finally, we investigate the remaining `CheckForRWPaths` property. As described earlier, we already discovered that DexProtector reads from `/proc/self/mounts`. Based on the property name, we assume that DexProtector additionally uses the read value to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

determine whether certain paths that are usually mounted as read-only are mounted as writeable. The popular open-source root detection library *RootBeer* [223] implements this approach as well and checks various system paths, such as `/system`. To confirm this assumption for DexProtector, we mounted several system directories – particularly those that are checked by RootBeer – as writeable. However, the `CheckForRWPaths` property of the passed JSON string was still set to `0`, leaving the described assumed behaviour unconfirmed.

5.2.2 LIAPP

All APKs used for the following analysis were generated on 16 August 2022 (LIAPP does not state an explicit version number).

Instead of a configuration file, the protection mechanisms that should be applied to an application are configured using the LIAPP web interface. However, the mechanisms "Source Code Encryption", "Anti-Debugging", and "Anti-Tamper" are enabled by default [166] and cannot be disabled with the LIAPP license we are using for analysis, as stated by Lockin company after further enquiry. After configuring the mechanisms and uploading the application, the (unprotected) APK is processed and the protected version can be downloaded afterwards.

General

First, we apply the tool with only the default/necessary options stated above.

Unpacking the resulting APK with *apktool* fails because of "invalid bytes" in one of the contained files. To circumvent this error, the `-s` flag of apktool can be used in order to prevent it from decoding sources. By specifying this flag, the APK is unpacked successfully, revealing the modified and added contents shown in Figure 5.5. Four shared libraries – one for each architecture (`arm`, `arm64`, `x86`, and `x86_64`) – with the name `libxbyabz.so` were added to the `lib` directory. Additionally, the file `LIAPP.ini` was added to the `assets` directory, containing plain text informing that the APK has been protected with LIAPP. Furthermore, the unpacked directory contains two DEX files – `classes.dex` as well as `classes2.dex`. As the DEX files are not decoded yet, we use *jadx* to disassemble and decompile them. While this process succeeds for `classes.dex`, it fails for `classes2.dex` due to "bad bytes", similar to before when apktool was applied initially. Therefore, we assume that `classes2.dex` is stored in an encrypted form inside the APK and decrypted as well as loaded during runtime.

Nonetheless, the decompiled code of the `classes.dex` file can be analysed. LIAPP added the package `com.liapp` to the APK. This package contains several obfuscated classes with identifiers containing non-printable characters. In addition, all strings were replaced by method calls with obfuscated strings as parameters returning the original strings, thus representing a form of string obfuscation. However, the deobfuscation method is directly implemented in one of the added classes and relies on simple bitwise XOR and AND instructions, allowing to reimplement the deobfuscation algorithm. In

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

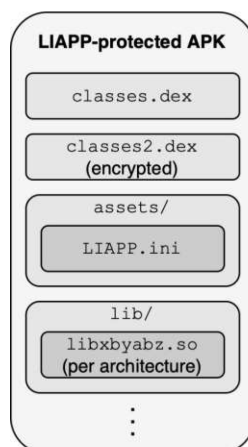


Figure 5.5: LIAPP: Contents of protected APK.

combination with a regular expression that identifies the obfuscated strings/deobfuscation method calls, all strings of the decompiled classes can be deobfuscated automatically, allowing a more efficient analysis of the added classes. One of the added classes is a custom `Application` class, which is also specified in the Android manifest and contains several functions for retrieving information about the device, such as the Android OS version and the Android application itself, such as its package name and granted permissions. In addition, the class is responsible for loading LIAPP's shared library. Furthermore, the custom `Application` class contains a single native method declaration that is called multiple times during initialisation. The other classes seem to be responsible for loading DEX files and their respective classes, as they leverage the `loadDex()` method of Android's `DexFile` [23] class used for loading DEX files during runtime. Additionally, the `ClassLoader` [17] class is used within several methods of the added classes in order to load classes of the loaded DEX file during runtime. In the case of the analysed APK, the `DexFile` and `ClassLoader` classes might be used for loading the `classes2.dex` file (after it has been decrypted) as well as its contained classes.

However, as one of the necessary options is "Anti-Debugging", attempts to hook the application using Frida fail, thus preventing a more detailed dynamic analysis. More precisely, the connection to Frida is disconnected within the native method that is defined in the custom `Application` class. Moreover, trying to circumvent this mechanism by overwriting the implementation of said method, causes the application to crash immediately due to some classes not being found. This leads to the conclusion that the previously described class decryption and loading routines are embedded within the method that also contains the anti-debugging/hooking mechanism, preventing overwriting the implementation without affecting the behaviour of the protected application. In addition, when trying to attach Frida to an already running application that is protected with LIAPP, the connection is terminated immediately, indicating the usage of `ptrace` [158] to attach to the own process in order to prevent debugging and hooking tools from

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

attaching. As the analysis of debug/hook detection mechanisms is not in the scope of this work, the matter of bypassing these mechanisms remains open for further research.

String Encryption

For analysing LIAPP's string encryption mechanism, the corresponding option in addition to the necessary ones was enabled.

Decompiling the protected APK using *jadx* shows that strings are replaced with different methods containing unprintable characters in their names. Every "decryption" method takes an integer as a single parameter and returns the original, decrypted string. Furthermore, the methods are defined in a single class (`com.liapp.y`) that is *not* present in the APK. One of the classes of the added `com.liapp` package contains a method that explicitly loads the class `com.liapp.y` using a `ClassLoader`. Thus, it is assumed that the class containing the "decryption" methods is contained in the additionally added DEX file and thus loaded during runtime, as described in the previous section. However, the corresponding method for decrypting the strings cannot be hooked without circumventing LIAPP's anti-hooking mechanism.

Class Encryption

For analysing LIAPP's class encryption mechanism, the "class protection" option in addition to the necessary ones was enabled. LIAPP allows specifying a filter for the classes that should be protected to avoid a drastic performance impact caused by protecting/encrypting a large amount of classes, such as the default Android API and embedded libraries.

Decompiling the protected APK using *jadx* shows that the classes of the evaluation application have been completely removed from the APK/`classes.dex` file. Instead, the `com.liapp` package that was added during the protection process file makes use of Android's `DexFile` [23] and `ClassLoader` [17] classes in order to load classes during runtime, as described in the "General" section. Furthermore, the `classes2.dex` file's size increased by approximately 4 kilobytes when using the class encryption mechanism. Thus, we expect that the protected classes where encrypted and stored within the `classes2.dex` file. This way, LIAPP does not need to perform additional operations for decrypting encrypted classes, as said DEX file is decrypted on the application's startup regardless of enabling the class encryption mechanism. However, similar to before, the methods for decrypting and loading the encrypted classes cannot be hooked without circumventing LIAPP's anti-hooking mechanism.

Root Detection

For analysing LIAPP's root detection mechanism, the corresponding option in addition to the necessary ones was enabled.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Decompiling the protected APK using *jadx* shows no change in functionality in comparison with the APK where only the necessary protection mechanisms have been applied to. Thus, we assume that the root checks are executed immediately after LIAPP's native library is loaded.

We further analyse the `strace` output upon the application's startup. Multiple `openat` [156] calls – used for attempting to open `su` binaries at different locations – can be observed. Additionally, files installed and required by Magisk, such as `/system/bin/magisk`, are checked. This way, the presence of potential superuser binaries and Magisk files can be determined by checking the return values of the `openat` calls. However, note that the same behaviour/system calls can be observed with only the necessary protection mechanisms enabled in LIAPP's web interface (i.e. root detection mechanism is disabled), leading to the conclusion that no additional functionality – except crashing the application on a positive check result – is added to the APK. On the physical Android device used for analysis, rooting is immediately detected and a corresponding dialogue is shown before the application crashes. Further, renaming the superuser binary on the physical device rooted through Magisk does not bypass LIAPP's root detection, as the existence of several Magisk-related files is checked.

In order to confirm this assumption, we execute the evaluation application on an `x86_64` Android emulator, running Android OS 11 without Google Play Services, causing the emulator to include an `su` binary. After renaming the binary, LIAPP still classifies the device as rooted, due to the device being detected as an emulator, as the dialogue depicted in Figure 5.6 implies. The shown message suggests that one or more system properties have been read in order to conclude that the executing device is an emulator. Modifying various potential related system properties reveals that merely the `ro.build.type` property, representing the build flavour of the OS image [15] [43], is compared against the string `"userdebug"`, which is the typical value on emulator images. Changing the `ro.build.type` property to any other, arbitrary value bypasses LIAPP's root detection mechanism (also see subsection 6.2.3).

In conclusion, LIAPP's root detection mechanism operates as follows:

1. Various potential locations of `su` binary are checked.
2. The property `ro.build.type` is checked against `"userdebug"`.
3. If the previous checks (1. and 2.) are negative, the existence of files required by Magisk is determined.

5.2.3 DashO

For analysing DashO we use version 11.2.1.

The mechanisms that should be applied are configured in a graphical user interface or directly in an XML file [207], as shown in Listing 5.9. In this example, only the string encryption mechanism is enabled.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

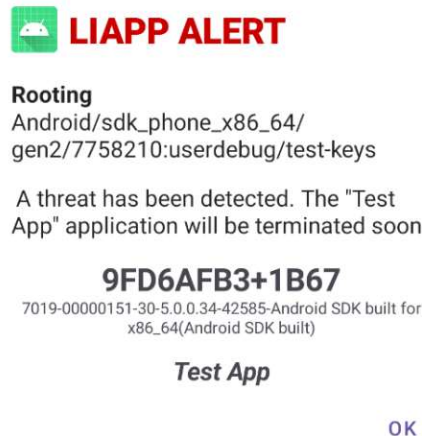


Figure 5.6: LIAPP: Dialogue indicating rooted device (emulator) was detected.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <dasho mode="android" version="11.2.1">
3     <stringencrypt option="on" />
4 </dasho>
```

Listing 5.9: DashO: Example configuration file.

General

We first apply the tool with an empty configuration, i.e. no protection mechanisms are applied, but DashO still processes the Android application.

Unpacking the resulting APK with *apktool* reveals no visible changes to the contents of the APK, i.e. no added classes or libraries. We further compute the differences of the bytecode (in smali representation). This comparison shows that there are no remarkable changes, except the insertion of `if` statements that check whether a certain time has passed and throw an exception in order to exit the application if this is the case. This behaviour is related to the usage of DashO's trial license, which allows using the tool and resulting applications for a short period of time. After the evaluation period expired, protected applications become unusable through the procedure described above.

String Encryption

DashO's string encryption mechanism allows configuring the "string encryption level" as well as the "number of decrypters" [207]. The former controls the strength of the used encryption algorithms. However, stronger algorithms reduce the performance of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the application as decryption at runtime consumes more time. The latter controls how many different methods used for decrypting encrypted strings are added to the application. For analysing the string encryption mechanism, we exclusively enabled string encryption (using the `<stringencrypt/>` element in the configuration file) and set both parameters to their default values of `2`.

Decompiling the protected APK with *jadx* reveals that strings were replaced by a sequence of newly generated method calls as shown in Listing 5.10. The generated methods are part of anonymous classes that were inserted into existing, randomly selected classes of the Java/Kotlin and Android APIs. For example, line 1 of Listing 5.10 calls a method of an inserted anonymous class of the `StringKt` class in order to generate a random integer. Based on this random number, one of the (in this case two) decrypter methods is selected to decrypt the encrypted strings (line 2). Note that in the case of the protected evaluation application analysed in this work, the selected decrypter method is always the same one, as the randomly generated number is multiplied by an integer and reduced modulo itself. The result is then compared to `0`, which always evaluates to `true`.

```
1  int subSequence3 = StringKt.AnonymousClass1.subSequence();
2  Log.i(..., StringKt.AnonymousClass1.subSequence((subSequence3 * 4) %
   ↪   subSequence3 == 0 ?
   ↪  "Jf}giceci/]p{}Uvb~npnb<jwkh!`vmibSqyo++io|x~5" :
   ↪  ForwardingListener.AnonymousClass1.split(56, "))4($3'.nxs"), 6
   ↪  ));
```

Listing 5.10: DashO: Replaced strings.

The anonymous class inserted into the `StringKt` class is shown in Listing 5.11. As explained previously, one method (lines 2 - 4) is used for generating a random integer, while the other one (lines 6 - 16) is used for the actual string decryption process. However, the decryption routine solely applies simple bit operations to each character, such as `XOR` and `AND`, allowing to reimplement the algorithm (see subsection 6.3.1).

In this example – as we have specified two decrypter methods in the DashO configuration – a second decryption method (`ForwardingListener.AnonymousClass1.split()`) is involved (see Listing 5.10). However, the implementation of this method only differs slightly to the one shown in Listing 5.11.

Root Detection

DashO allows developers to configure various "responses" for its root detection mechanism and environment checks in general [207]. For example, DashO can be configured to throw an exception or cause the application to "freeze" if a rooted device was detected. In addition, the specified responses can be configured to occur only with a specified probability. As a result, the observed behaviour is not deterministic when trying to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```java
public class AnonymousClass1 {
    public static int subSequence() {
        return new Random().nextInt(75) + 1;
    }

    public static String subSequence(String str, int i) {
        char[] charArray = str.toCharArray();
        int length = charArray.length;
        int i2 = 0;
        while (i2 != length) {
            charArray[i2] = (char) (charArray[i2] ^ (i & 95));
            i++;
            i2++;
        }
        return String.valueOf(charArray, 0, length);
    }
}
```

Listing 5.11: DashO: Inserted anonymous class used for decrypting strings.

analyse a protected application. Additionally, DashO can be configured to inject the environment checks into various methods within the source code.

For the purpose of analysing the functionality of the root detection mechanism itself, we configure DashO to inject the root detection functionality into a single method and call a method for setting the result of the check – without taking advantage of any additionally configurable behaviour, such as throwing an exception. The configuration for achieving this behaviour is shown in Listing 5.12. Line 5 specifies the field or method that is called to store the result of the check. Further, the class and method where the detection functionality should be injected into is specified in lines 7 and 8.

Decompiling the protected APK with *jadx* reveals that the root detection functionality was inserted into the specified `performCheck()` method. However, the implementation is obfuscated through reflection and string encryption. Thus, method calls are not directly visible. Nonetheless, DashO applies the same procedure for encrypting its own strings as when explicitly enabling the string encryption mechanism as described previously. As a result, the same approach for decrypting strings can be used (see subsection 6.3.1). In combination with the usage of a regular expression in order to identify encrypted strings, all strings of (decompiled) class files can be decrypted automatically. Applying this procedure reveals the methods that are called through reflection and used for detecting a rooted device. The decompiled and "string-decrypted" source code reveals that it fulfils two primary purposes: Checking whether certain files, such as `su` binaries, exist by using several `File.exists()` [25] calls, and checking whether certain applications, such as root management applications, are installed on the device by leveraging Android's

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dasho mode="android" version="11.2.1">
    <injection>
        <checks>
            <rootCheck action="setRooted()">
                <locations>
                    <classes name="a.b.app.DashORootCheck">
                        <method name="performCheck" signature="" />
                    </classes>
                </locations>
            </rootCheck>
        </checks>
    </injection>
</dasho>
```

Listing 5.12: DashO: Root detection configuration.

PackageManager [31]. A snippet of the decompiled code containing the former purpose is shown in Listing 5.13. In lines 4 - 8 an array is filled with various strings representing different locations where a superuser binary could be present. The injected code then proceeds to iterate through this array, checking whether the file with the respective path exists (lines 10 - 19). If this is the case, a variable is set to some value (line 14), which is used later on to check whether any indication for a rooted device has been found (line 23). In addition to the decompiled code, strace shows several faccessat [152] calls to check whether files exist, confirming the described behaviour.

Between checking potential superuser binaries and the final assignment of the check's result, the seconds mechanic of the root check – namely checking whether certain applications are installed on the device – is performed (not depicted in Listing 5.13). First, several package names are stored in an array. Afterwards, the array is iterated and the getPackageInfo() method of the PackageManager class is used to check whether the respective package is installed on the device. Again, if this is the case, a variable is set to some value, indicating that a rooted device was detected. In addition to the decompiled code, jtrace confirms the usage of Android's PackageManager for checking installed packages.

On top of the primary checks described above, the existence of /system/etc/security/otacerts.zip is checked. This file is used to verify the signature of over-the-air (OTA) updates of the Android OS [48]. A missing file indicates that a custom Android OS image that is not updated through OTA updates is installed. Furthermore, it is checked whether the Build.TAGS constant [15] (corresponding to the ro.build.tags system property) is equal to test-keys, which indicates an emulator (i.e. developer build) [234].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  private final void performCheck() {
2      // ...
3
4      String[] strArr = new String[10];
5      strArr[0] = "/system/bin/su";
6      strArr[1] = "/system/bin/sudo";
7      // ...
8      strArr[9] = "/vendor/bin/sudo";
9      int i4 = 0;
10     for (int i5 = 10; i4 < i5; i5 = 10) {
11         File file = new File(strArr[i4]);
12         Class<?> cls = Class.forName("java.io.File");
13         if (((Boolean) cls.getMethod("exists").invoke(file)).
   ↪   booleanValue()) {
14             d = 100.125d;
15             break;
16         } else {
17             i4++;
18         }
19     }
20
21     // Checking whether certain applications are installed...
22
23     if (d < 100.0d) {
24         setRooted(false);
25         return;
26     }
27     setRooted(true);
28 }
```

Listing 5.13: DashO: Snippet of root detection implementation.

5.3 Main Differences Between Analysed Tools and Mechanisms

Based on the results of the previous analysis, the following section describes several differences in the implementations of anti-reverse engineering tools and mechanisms.

5.3.1 Native vs Bytecode Implementation

Some mechanisms offered by anti-reversing tools, such as environment checks and string encryption, are suited to be implemented in native code, which is compiled to machine code and stored inside an APK as a shared object. Implementing such mechanisms in native code makes reverse engineering more cumbersome, as machine code reveals less information when decompiled compared to bytecode [88]. Additionally, native code

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

implementations can be obfuscated using well studied, advanced obfuscation techniques, thus further impeding reverse engineering [114]. For example, packing techniques can be employed in order to unpack and load code during runtime, making static analysis even more difficult.

In our analysis, we observed that DexProtector pursues the native approach. DexProtector's native library, which is stored in an encrypted form and additionally employs packing techniques, implements the offered environment checks and decryption routines for the string and class encryption mechanisms.

In contrast, DashO directly injects the implementations of the environment checks into the specified Java methods. Although the injected code is obfuscated through reflection and string encryption, it is reversible and implementation details can be revealed. Similarly, DashO injects the decryption routines for its string encryption mechanism directly into Java classes, allowing to reimplement them and thus restore the original strings.

LIAPP provides a combination of the two approaches. While its environment checks are deeply embedded into the initialisation routine of the native library, impeding overwriting the implementations directly, the string decryption routine is implemented as bytecode. However, the corresponding code segments for decrypting strings are not available when statically analysing the APK, as they are loaded dynamically during runtime.

5.3.2 Mandatory Hooking Detection

For analysing the mechanisms that are focus of this work, we aimed to exclusively enable the mechanisms in question. However, out of the three analysed tools, one of them, namely LIAPP, prevents some "default" options, with one of them being the inclusion of hooking detection, to be turned off. As the hooking detection mechanism prevents dynamic analysis of the application with tools like Frida, the mandatory hooking detection does not only protect the application where LIAPP was applied to but also LIAPP and its applied mechanisms.

In comparison, DexProtector and DashO do not force their hooking detection mechanisms to be enabled. Instead, the mechanism can be enabled or disabled in the respective configuration file where other environment checks, such as root detection, are configured as well. This allows dynamically analysing other anti-reversing mechanisms provided by the tools without the need of bypassing a hooking detection mechanism.

5.3.3 Entry Points of Environment Checks

The three analysed tools differ in the way the environment checks are called. Besides the usability aspect from an application developer's perspective, single entry points make it easier for reverse engineers to analyse and bypass the checks.

DexProtector only provides a single (native) method per environment check that is called to perform the respective check. Additionally, two callback methods can be configured, with one of them being called based on the check's result. Overwriting either the method

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

that invokes the environment check or the one that is called on a positive check result allows bypassing the check.

DashO allows configuring multiple methods that perform an environment check and multiple methods/actions that are called/performed for handling the result of the check. Thus, it might not be sufficient to intercept a single method call that performs an environment check, as there could be multiple entry points present.

LIAPP takes a different approach. Instead of injecting calls to methods that perform the environment checks, the checks are already embedded into the loading routine of the native library. Hence, no direct entry point is visible and the trivial approach of overwriting the loading routine to bypass the hooking detection fails, as it impedes the correct functionality of the application.

5.3.4 Complexity of Root Detection

In addition to the fact that environment checks – in particular root detection as we analysed it as part of this work – can be implemented in native or Java code, the complexity of the mechanisms' implementations varies.

DexProtector performs several separate checks, ranging from checking the existence of superuser binaries to analysing `/proc/self/mounts`, for detecting a rooted device. Should one of these check results turn out to be positive, the device is classified as rooted.

LIAPP and DashO rely on less indicators. Both tools check various potential locations of the `su` binary as well as the value of a system property. While LIAPP additionally checks the existence of Magisk-related files, DashO determines whether a root management application, such as Magisk, is installed. Further, DashO checks the existence of certificates used for over-the-air updates.

5.3.5 Obfuscation of Tool-Related Code

All of the analysed tools aim to conceal implementation details by applying various obfuscation techniques, which vary from tool to tool, to the injected bytecode.

DexProtector and LIAPP apply identifier renaming, control flow obfuscation, as well as string encryption to their respective injected classes, such as the `Application` class. Furthermore, some tool-related functionality, such as DexProtector's TLS certificate pinning mechanism or LIAPP's string decryption routine, is loaded during runtime.

DashO applies more rudimentary obfuscation techniques, namely string encryption and obfuscation through reflection, to its injected functionality. In addition, the injected code segments are directly present in the APK, allowing to statically analyse the injected functionality.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Bypass of Anti-Reverse Engineering Mechanisms

The following chapter describes how anti-reverse engineering mechanisms provided by the analysed tools can be bypassed by utilising the knowledge of internal implementation details gained through the previous analysis. Further, we present a basic approach for automatically identifying the applied anti-reverse engineering tool in order to provide a solution for bypassing the anti-reversing mechanisms applied by any of the three tools that are focus of this work.

6.1 DexProtector

In the following, we describe several approaches to revert/bypass DexProtector's [145] class and string encryption, TLS certificate pinning, as well as root detection mechanisms. We implemented these approaches as Frida [95] scripts and tested them by applying them to our evaluation application protected with DexProtector version 12.0.1.

6.1.1 String Encryption

In order to revert DexProtector's string encryption mechanism to reveal all original, decrypted strings, we present two approaches.

First, the native `s` function that takes an encrypted string and returns the decrypted one can be hooked in order to track the function calls with the passed parameters and corresponding return values. This procedure can be realised by hooking the JNI function `RegisterNatives` [195], which is used to register native functions during runtime. By hooking this function, the address of the `s` function can retrieved, allowing to hook the function using Frida's Interceptor API [96]. Whenever the `s` function is called, the Frida script converts the passed string parameter, i.e. the encrypted string, into hexadecimal

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

representation as it might contain unprintable characters and logs it. Before the function returns, the Frida script logs the returned value, i.e. the corresponding decrypted string. Furthermore, we extended the script to store the mappings of encrypted to decrypted strings in a file, allowing further processing, e.g. automatically replacing the encrypted strings within a decompiled class with its decrypted counterparts.

The second approach consists of dumping the memory of the application during runtime. As the strings must be decrypted before usage, i.e. the `s` function has to be called with the corresponding parameter, a memory dump of the application usually contains the strings that have been loaded up to the point in time when the memory was retrieved. Various methods for extracting the memory of an Android application exist. For example, the command line tool *Android Debug Bridge (adb)* [7], which enables communication with an Android device, can be used to create a heap dump of a specified application. Afterwards, the resulting dump can be opened with the *Android Profiler* [39], which is directly integrated into *Android Studio* [9]. This method provides a very clearly represented overview of all Java instances (including strings) stored on the heap. However, Android does only allow dumping the heap of applications that are built with the `android:debuggable` attribute set to `true` inside the Android manifest. Therefore, already built release applications have to be set to "debuggable" through repackaging and resigning.

In addition, memory dumps can be created using various (open-source) projects developed for this purpose. For example, *fridump* [105] leverages Frida in order to extract the memory of an application. The memory contents are written to several files, which can be used for further analysis and/or filtering, e.g. only including all printable strings.

6.1.2 Class Encryption

Similar to the string encryption mechanism, encrypted classes must be decrypted during runtime in order to be executed. Thus, DexProtector's class encryption mechanism can be reverted by extracting the decrypted classes during runtime in order to analyse them further, e.g. through decompilation. To do so, we present two approaches.

As a first approach, the `DexFileLoader::OpenCommon` [4] function, which is used to load a DEX file during runtime, can be hooked in order to extract the DEX file at the address passed as a parameter. Further, the total size of the DEX file (in bytes) is passed as another parameter and is also included within the header of the DEX file [47]. Based on the address and size, the DEX file can be extracted from memory using Frida. In addition to hooking `OpenCommon`, the `ClassLinker::DefineClass` [3] function has to be hooked in the same way as described above, as this function is used to load specific classes from DEX files. However, when applying this approach to our evaluation application, inspection of the dumped files revealed that the extracted DEX files are valid but not complete, i.e. parts of the application's bytecode are missing. This observation leads to the conclusion that DexProtector splits up the DEX file(s) that is/are loaded during runtime and distributes loading the subfiles across various functions, such as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

OpenCommon and DefineClass. As there could be additional functions used to load DEX files, extracted DEX file(s) might not be complete.

To circumvent the problem described above, another approach, consisting of extracting DEX files from the application's memory based on the DEX file structure [47], can be applied. The open-source project *frida-dexdump* [127] implements this approach leveraging Frida to identify and dump the corresponding memory sections. In order to identify DEX files within an application's memory, two criteria are used: First, DEX files must start with the bytes 64 65 78 0a ("dex\n"), followed by a version number. Second, the map_list element [47], a list of the entire contents of the DEX file, must exist and be valid. frida-dexdump enumerates all readable memory segments and checks both conditions for each of the segments. If at least one of these two conditions is fulfilled, the relevant memory section is dumped after retrieving the size of the DEX file by reading the corresponding header field or, if the header is not or only partly present, by subtracting the address of the map_list ending with the starting address.

6.1.3 TLS Certificate Pinning

Based on the analysis results, bypassing DexProtector's TLS certificate pinning mechanism is possible through overwriting the implementation of the method that replaced the previous url.openConnection() call and performs further method calls in order to invoke the certificate verification process. By returning url.openConnection(), we prevent further verification logic from being executed and thus are able to bypass the pinning mechanism. However, the class and name of the target method have to be determined first, as class and method name are generated randomly. Therefore, we hook the ClassLinker::DefineClass [3] function, which is used to load classes during runtime. For each loaded class, we check if it contains a method with java.net.URL [41] as a single parameter. If this is the case, the implementation of the found method has to be modified as described above in order to circumvent the TLS certificate pinning routine.

6.1.4 Root Detection

DexProtector's root detection mechanism can be bypassed through a variety of ways. First, by hooking the RegisterNatives [195] function, which is used to register native functions during runtime, the registered native functions can be retrieved. One of the registered functions is the r() function, which was inserted into the specified doProbe() method and used to execute the actual root detection mechanism within DexProtector's native library. Overwriting the doProbe() or r() function with an empty implementation causes the root check to not being executed, resulting in none of the specified callbacks being called. If this behaviour is not desired, one of the two functions can be overwritten with an implementation that calls the callback used for indicating a negative root check result. Alternatively, the positive check callback can be overwritten such that it calls the negative check callback. In addition, in order to cause

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

as little side effects as possible, the JSON object that is passed as a parameter to one of the callbacks should be retrieved and modified accordingly, i.e. setting all flags to `0`, before passing it to the negative check callback.

6.2 LIAPP

In the following, we present potential approaches to bypassing LIAPP's [165] string encryption, class encryption, and root detection mechanisms. However, due to LIAPP's necessary hooking/Frida detection, we could not evaluate the effectiveness of those approaches. Nonetheless, we additionally provide an approach for bypassing LIAPP's root detection mechanism that does not leverage Frida or another dynamic code instrumentation/hooking tool, and instead relies on modifying files on an Android emulator. The latter approach has been tested on the basis of our evaluation application protected with LIAPP on 16 August 2022.

6.2.1 String Encryption

LIAPP's string encryption mechanism depends on a method that takes an integer as a parameter and returns the decrypted string. Although this decryption method is not present in the APK and loaded during runtime, the method can potentially be retrieved by hooking the `ClassLinker::DefineClass` [3] function. This approach is similar to the one used for identifying the relevant method in order to bypass DexProtector's TLS certificate pinning mechanism. Hooking the loaded decryption method and logging the parameter as well as return value should reveal the mappings of integers to strings.

Additionally, dumping the application's memory, as described in subsection 6.1.1, could succeed in dumping the decrypted strings, as the strings must be decrypted before usage.

6.2.2 Class Encryption

For bypassing LIAPP's class encryption mechanism, two approaches, similar to the ones used for reverting DexProtector's class encryption mechanism, could be successful. On the one hand, as LIAPP leverages the `ClassLoader` [17] API in order to load classes during runtime, the corresponding functions, such as `OpenCommon` [4] and `DefineClass` [3], can be hooked. On the other hand, the approach consisting of dumping the application's memory and identifying as well as extracting present DEX files based on the structure of DEX files [47] can be applied. Both of these approaches are the same as for DexProtector, see subsection 6.1.2.

6.2.3 Root Detection

Tracing LIAPP's performed system calls revealed that `openat` [156] is called several times to determine whether a superuser binary or Magisk-related [241] files exist in different possible locations. Therefore, using Frida's Interceptor [96] to hook the corresponding

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

open function could potentially succeed in bypassing the root detection mechanism. In doing so, the passed path is examined and – if it contains su or magisk – the return value is replaced by -1, simulating the respective file not being present. Note that we neglect the check of the ro.build.type property for this approach, as we assume that the described procedure is performed on a physical device.

In addition, we could confirm the success of a Frida-less approach using an x86_64 Android emulator, running Android OS 11 without Google Play Services, thus including a superuser binary and behaving similar to a rooted, physical device. The approach consists of trivially renaming the su binary, which bypasses the first check. As Magisk-related files are not present on the emulator, no further action has to be taken in that regard. However, the ro.build.type property, which is equal to "userdebug", has to be modified. To do so, the Android emulator has to be booted with a writeable system partition. Afterwards, the corresponding line in the /system/build.prop file containing all system properties, can be modified. While the typical other values of the ro.build.type property are "user" or "eng" [43], the property can be changed to an arbitrary value other than "userdebug". After a reboot of the emulator, the modification can be confirmed using the getprop [42] command, which lists all present system properties. If the modification was successful, LIAPP no longer classifies the device/emulator as rooted.

6.3 DashO

In the following, we describe our approaches for reverting/bypassing DashO's [205] string encryption and root detection mechanisms. We have implemented the approaches as Frida scripts and tested them on the basis of our evaluation application protected with DashO version 11.2.1.

6.3.1 String Encryption

Due to DashO's string decryption routine only relying on simple bit operations, it can be reimplemented in other languages, such as Python [211]. As an example, Listing 6.1 shows a Python implementation of DashO's string decryption routine that was revealed during the analysis phase (see Listing 5.11). After reimplementing the decryption routine, further, automatic processing can be performed. For example, using regular expressions, calls of the decryption functions within decompiled code can be identified and replaced by the respective decrypted strings that are obtained by calling the reimplemented string decryption routine.

In addition to statically decrypting strings, a dynamic approach can be used. After identifying the decryption functions, e.g. through static analysis, the identified functions can be hooked in order to monitor the passed parameter (encrypted string) and corresponding return value (decrypted string).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```python
def decrypt(encrypted, i):
    arr = list(encrypted)
    i2 = 0
    while i2 != len(encrypted):
        arr[i2] = chr(ord(arr[i2]) ^ (i & 95))
        i += 1
        i2 += 1
    return ''.join(arr)
```

Listing 6.1: Python implementation of DashO's string decryption routine.

6.3.2 Root Detection

DashO's root detection mechanism consists of two primary checks – determining whether certain files exist and checking whether certain applications are installed. For bypassing the former measure, the corresponding `File.exists()` [25] calls can be overwritten in order to change the return value to `false`, in case a root-indicating file is checked. Alternatively, the return value of the underlying native function `faccessat` [152] (which calls the identically named system call, as revealed through system call tracing) can be modified. For bypassing the latter measure, the `PackageManager.getPackageInfo()` is hooked and the passed package name is replaced with a non-existent one in case the method was called with a package name indicating a rooted device. Afterwards, the original implementation of the method is called with the original or modified package name in order to provide an appropriate return value. We assume that the described procedure/Frida script is executed on a physical device running an unmodified Android version, which allows neglecting the two additional root check measures, namely the comparison of the `Build.TAGS` [15] constant as well as checking the existence of `/system/etc/security/otacerts.zip`.

6.4 Automatic Identification of Applied Anti-Reversing Tool

For identifying which of the analysed anti-reversing tools has been applied to a given application, we present two techniques. On the one hand, a static approach, implemented by the tool *APKiD* [217], can be used. On the other hand, a dynamic approach that we implement as a Frida [95] script, can be applied. Regardless of the used approach, the output of the "identification process" can be used in order to execute the bypassing mechanisms/scripts depending on the identified tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.4.1 Static Approach – APKiD

APKiD [217] – named after the popular tool *PEiD* [1], which provides a similar functionality for Portable Executable (PE) files [181] – aims to identify the compiler, packer, and/or obfuscator that was applied to an APK. The tool unpacks the given APK and applies *YARA* [251] rules to the APK (e.g. in order to determine whether certain files are present), potential native binaries, and the DEX bytecode. YARA was initially developed to help identifying malware samples through specified descriptions based on textual or binary patterns [251]. In the case of APKiD, YARA is used to describe the characteristics of various obfuscation and anti-reverse engineering tools in order to identify them. APKiD implements YARA rules for two out of the three tools that were analysed in this work – DexProtector and LIAPP. For identifying DashO, we implement a custom YARA rule that is applied to the DEX bytecode. In the following, we will briefly describe the YARA rules that are used to identify DexProtector, LIAPP, and DashO:

- *DexProtector*: As DexProtector stores its (encrypted) native libraries inside the `assets` directory, a YARA rule checking the existence of such is capable of identifying DexProtector.
- *LIAPP*: For identifying LIAPP, a YARA rule that checks the existence of the `LIAPP.ini` file that has been added to the `assets` directory is used.
- *DashO*: As DashO does not add native libraries or other files to the APK, a different approach than determining the existence of such files must be pursued. DashO injects additional code segments, e.g. for performing various environment checks or decrypting strings during runtime, depending on the configuration. The methods for decrypting strings are also included when the string encryption mechanism is not enabled, as DashO's injected code takes advantage of its own string encryption mechanism in order to hide the intended functionality. Additionally, the executed statements always follow a specific pattern: First, a random number is generated. The generated number is then used as part of a multiplication and modulo operation. The result is then used within an if condition to determine which decryption function is finally called. Therefore, the corresponding bytecode [46] of the instruction sequence can be used within a YARA rule that determines whether the DEX file(s) of an APK contain(s) the specified byte sequence.

6.4.2 Dynamic Approach – Frida

In addition to statically analysing an APK, Frida [95] can be used to reveal operations that are performed during runtime and allow inferring the applied anti-reverse engineering tool. For example, the `System.loadLibrary()` function [38] that is called to load native libraries can be hooked in order to determine the passed parameter, i.e. the file name of the library. As the name of the loaded libraries are fixed and known beforehand, the applied anti-reversing tool can be determined through comparing the parameter of the `System.loadLibrary()` call against pre-defined values. In particular, our developed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Frida script determines whether the library name is equal to `"dexprotector"` or `"xbyabz"`, in order to identify DexProtector and LIAPP, respectively.

Alternatively, as an immediate execution of a bypassing script might be necessary but the `System.loadLibrary()` call might be delayed, Frida can be used to enumerate the present classes during runtime. If certain tool-specific classes are present, the applied tool can be implied. In addition to the enumeration of all loaded classes, Frida's `Java.use()` function [96] may be used to attempt retrieving a wrapper for the given class, provided that the exact class (and package) name is known beforehand. If no exception is thrown, it can be inferred that the specified class exists. In the case of DexProtector, the presence of the injected `AppComponentFactoryDP` and `MessageGuardException` classes implies the usage of said tool. Additionally, in contrast to the presented static approach using APKiD, dynamically enumerating classes also reveals classes that are loaded during runtime, which might be the case when certain mechanisms, such as class encryption, are applied. This approach is also necessary for detecting tools that do not rely on native libraries, such as DashO. More specifically, the presence of injected anonymous classes that contain the methods used for decrypting strings during runtime, can be determined for identifying DashO. As there are several possible classes where DashO injects its anonymous classes, the existence of various classes, such as `androidx.core.text.StringKt$1` (`$1` indicates an anonymous class), has to be checked using the previously described approach leveraging Frida's `Java.use()` function. The whole described dynamic approach, which allows determining if DexProtector, LIAPP, or DashO has been applied to an application, is depicted in Figure 6.1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

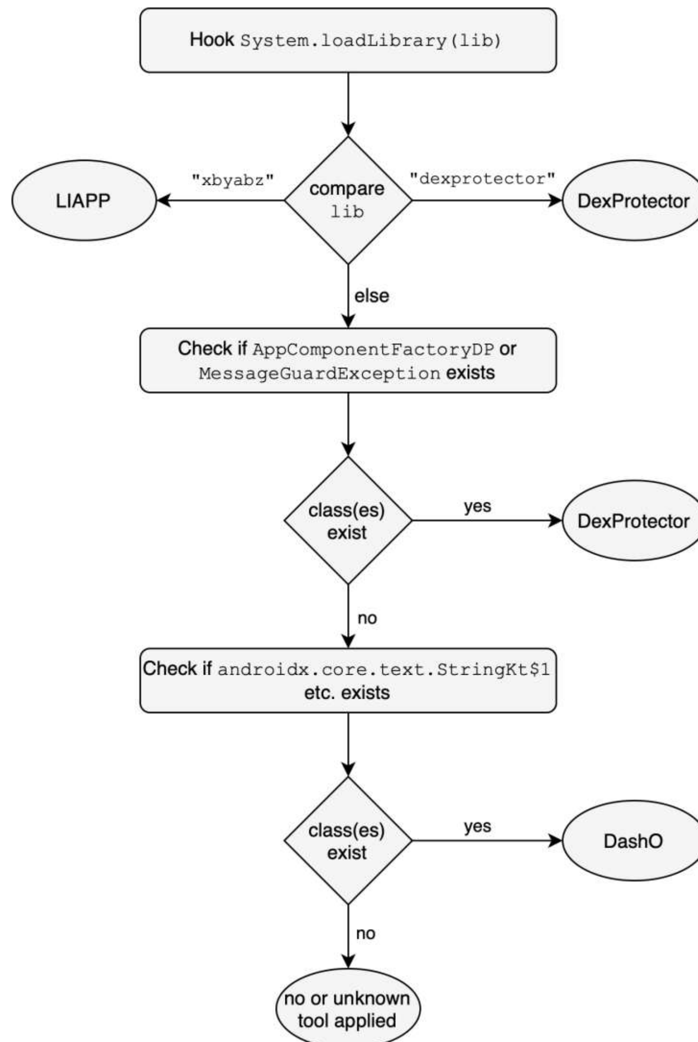


Figure 6.1: Dynamic approach for identifying applied anti-reversing tool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Possible Improvements of Anti-Reverse Engineering Mechanisms

Based on the previous analysis results and bypassing strategies, this chapter presents and evaluates possible approaches for improving the analysed anti-reverse engineering mechanisms and tools.

7.1 Possible Improvements

This section presents several possible improvements for improving the analysed anti-reverse engineering mechanisms and tools.

7.1.1 I1: Prioritise Debugging and Hooking Detection

In order to analyse (and bypass) various anti-reversing mechanisms provided by anti-reverse engineering tools, this work leveraged dynamic analysis, especially function hooking, to a large extent. Although each of the analysed tools provides the possibility to detect whether a hooking framework is attached, two out of the three analysed tools do not react to the detection's result accordingly. More specifically, in the case of *DexProtector* [145] as well as *DashO* [205], the environment checks are executed separately from each other and do not influence the behaviour of the protected application. In particular, the usage of a debugging or hooking tool, such as Frida, might have been detected, but further measures, such as preventing further execution of the application, are not taken. Such reactions have to be explicitly configured/implemented by the application developer. As a result, implementation details of the "protected" application

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

along with the anti-reversing tool can be revealed through dynamic approaches, even though the usage of such tools might have already been detected.

In the interest of circumventing the described issue, we suggest to prioritise debugging and hooking detection mechanisms in order to stop the execution of an application in case the usage of such tools has been detected. Ideally, these checks are deeply integrated into the loading procedure of the application, causing the application to immediately terminate in case debugging or hooking tools are detected, thus preventing further analysis during runtime.

Moreover, the described strategy can be used to prevent the execution of decryption routines. In particular, anti-reverse engineering tools could aim to avoid executing the decryption routines of corresponding mechanisms immediately at the start of the application. Instead, the decryption of strings, classes, etc. could be performed after various environment checks have been executed and determined a “safe” environment, i.e. no debugging or hooking tools are attached. Therefore, decryption routines are not executed when analysis tools are used, preventing the extraction of decrypted contents.

7.1.2 I2: Prevent Debugging and Hooking Tools From Attaching

In addition to detecting potentially attached debugging and hooking tools, mobile application anti-reversing tools can aim to prevent reverse engineering tools from attaching in the first place. A basic approach for doing so consists of forking a child process and attaching it to the parent via `ptrace` [158]. As only one tracer per process is allowed, further attempts to attach to the parent process will fail [198]. Although a rudimentary implementation of this approach can be bypassed easily, e.g. through killing the child process and therefore also the tracer of the parent process, more sophisticated techniques, such as forking multiple processes tracing each other, can be employed [198].

7.1.3 I3: Run Environment Checks Repeatedly

Executing various environment checks is one of the essential features of the majority of mobile anti-reverse engineering tools. However, during our analysis, we observed that some protection tools only execute these checks at specific points during the runtime of an application, e.g. at the start of the application. With such centralised approaches, entry points to various environment checks are relatively straight-forward to identify for reverse engineers, allowing to remove or overwrite the specific function calls. Further, time-of-check to time-of-use problems when attaching debuggers or hooking tools during the runtime of an application *after* the corresponding checks have already been executed and the (negative) check results have been evaluated in order to evade detection might arise.

To circumvent the described scenario, anti-reversing tools can aim to avoid executing environment checks at one single point during the application’s runtime. Instead, the environment check logic can be executed at different points in time of the application’s runtime. On the one hand, this can be achieved by inserting multiple calls to the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

check routines at different locations within the application's code base. On the other hand, anti-reverse engineering tools might inject calls to environment check routines into commonly used Android/Java APIs, making it difficult to trace the relevant code sections. Taking it one step further, anti-reversing tools could implement periodic executions of environment checks. Ideally, a native library that starts a routine for repeatedly executing the mechanisms' implementations is used.

7.1.4 I4: Avoid Usage of C Standard Library Functions

Environment checks often rely on system calls to perform the required checks, as our analysis shows. For example, root detection implementations might leverage `open` [156] or `stat` [159] to determine whether certain files exist [234]. Typically, system calls are not invoked directly, but by corresponding wrapper functions contained in the C standard library (libc) [155] [153]. However, as libc is a dynamic library, it exports the system call wrapper functions, allowing to dynamically instrument, e.g. hook, the functions directly [81]. As a result, the executed operations used for providing anti-reversing mechanisms, such as environment checks, can be intercepted and overwritten directly.

In order to mitigate the described attack vector, anti-reversing mechanisms could refrain from using libc functions for essential operations. Instead, system calls can be called directly. This task is not trivial, as libc functions usually perform other required steps, such as copying arguments to the appropriate registers, in addition to invoking the corresponding system call [153]. Thus, *musl* [184], an implementation of libc built on top of the Linux system calls API and optimised for static linking, can be used. Although system calls can still be traced/intercepted using dynamic code instrumentation, e.g. by leveraging Frida's Stalker API [97] allowing to trace executed instructions, the process of identifying and intercepting the essential operations is made more cumbersome compared to when wrapper functions of the C standard library are used.

7.1.5 I5: Introduce Indeterminism to Environment Checks

Analysing applications often represents a trial and error procedure, consisting of aiming to verify assumptions through repeatedly executing the application in question and observing the outcome and executed operations. Commonly, a deterministic behaviour is assumed, allowing to conduct the described procedure. Therefore, indeterminism can result in an impediment during analysis.

In the case of mobile anti-reversing tools, indeterminism can be implemented through a variety of ways. For example, *DashO* allows configuring different behaviours, such as terminating or freezing the application in case of a positive check result [207]. The actual carried out behaviour is selected randomly. However, this approach only consists of randomising the performed actions *after* the check routines have been executed and relies on the developer to be configured accordingly. Thus, we suggest introducing indeterminism within the implementations of anti-reversing mechanisms themselves. For example, the execution order of the performed environment checks during initialisation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

can be randomised on each application run. Further, the instructions of each environment check can be executed in different sequences, wherever possible. Finally, in case of a positive check result, an approach similar to the one DashO applies might be used to ensure that it is not possible to immediately imply the check result, e.g. by means of a terminating application. Instead, reactions that differ on each application run could be performed.

7.1.6 I6: Verify Execution of Environment Check Routines

Many bypassing strategies for mechanisms provided by anti-reversing tools rely on overwriting specific functions that would otherwise perform various checks and result in the application's behaviour being changed drastically, e.g. by terminating the application. Performing such bypassing strategies often remains undetected, which is commonly the case when environment check routines are executed without being interwoven with other parts of the application.

To counteract, anti-reversing tools can aim to ensure that/verify whether environment check routines have been executed completely. An example for this approach has been observed during the analysis of *LIAPP* [165]. LIAPP interweaves its hooking and debugging detection mechanisms with the loading/initialisation procedure of its native library that is required for a proper functionality of the protected application as it implements other required mechanisms, such as the class decryption routines. Therefore, the initialisation of the native library entails the execution of required environment checks, such as hooking detection. Thus, as these checks are integrated into the initialisation process, trivial bypassing strategies, such as overwriting the executed functions used for initialisation and therefore also performing required environment checks, will result in the application terminating, as the initialisation procedure was interfered with.

In addition to embedding environment checks into essential initialisation procedures, implementations of anti-reverse engineering mechanisms might aim to verify whether the intended functions have been executed completely. A fundamental approach for doing so is to set one or multiple flags as part of the execution of the required checks. Later on, during the runtime of the protected application, these flags can be checked at several locations, and enforce immediate termination of the application in case one of the flags was not set, indicating that one or more required checks have not been executed properly. However, as debuggers and hooking frameworks could be used to modify the values of the flags, such techniques should generally be implemented in combination with (advanced) obfuscation techniques, for which we describe some examples in the following section.

7.1.7 I7: Implement Advanced Obfuscation Techniques

Anti-reversing tools commonly employ obfuscation techniques for two main reasons: On the one hand, some tools support obfuscating the application code as part of the offered anti-reversing mechanisms. On the other hand, most tools focus on obfuscating the injected routines that implement various mechanisms, such as environment checks,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

themselves. During our analysis, we observed that – although obfuscation was applied to the injected routines – the used obfuscation techniques were mostly restricted to standard techniques, such as string encryption or obfuscation through reflection. As a result, we were able to partly reconstruct the obfuscated code through restoring the original strings in order to gain knowledge about implementation details of the implemented mechanisms.

To counteract, anti-reversing tools can aim to not only rely on standard obfuscation techniques, but instead leverage advanced obfuscation techniques. For, example *Mixed Boolean-Arithmetics (MBAs)* [267] represent a way to transform simple expressions into representations that use arithmetic and boolean operators and are difficult to analyse [163]. Although the transformation changes an expression's complexity drastically, the semantics of the original expression is preserved. An example for an expression that was obfuscated using an MBA transformation is shown in Listing 7.1, where the expression x+y was transformed into a complex expression using both arithmetic and boolean operations with an additional variable z that has no effect on the result [163].

```
1   int fun(int x, int y, int z) {
2       int c;
3       c = x+y;
4
5
6
7
8
9       return c;
10  }
```

```
1   int fun(int x, int y, int z) {
2       int c;
3       c = 4*(~x&y)-(x^y)-(x|y)+4
    ↪   *~(x|y)-~(x^y)-~y-(x|~y)+1
    ↪   +6*x+5*~z+(~(x^z))-(x|z)-2
    ↪   *~x-4*(~(x|z))-4*(x&~z)+3*
    ↪   (~(x|~z));
4
5       return c;
6   }
```

Listing 7.1: MBA transformation example [163].

Another technique aiming to increase the complexity of a program's control flow are *opaque predicates* [72] [73]. Opaque predicates can be applied for different purposes, such as software watermarking [185] [51] or obfuscation [268] [139]. Typically, opaque predicates refer to some constant values that are known during build time but difficult to reveal during analysis, especially when combined with other obfuscation techniques [259]. In order to translate this concept for obfuscation purposes, constants can be replaced with methods that contain complex expressions (potentially also MBAs [163]) and always return the same value, regardless of the passed parameter(s).

7.2 Expert Evaluation

For evaluating the presented improvements, we interviewed 5 experts in the field of IT security. After some introductory questions related to the experts' opinions on and experiences with anti-reversing mechanisms, the possible improvements were presented

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

one after the other. For each of the presented possible improvements, the experts were asked to estimate the effectiveness (regarding the impediment of bypassing anti-reversing mechanisms and application analysis) of the given possible improvement on a scale from 1 to 5 (1 = not effective at all, 5 = very effective) and to justify their estimations in detail. The experts were asked to consider the presented improvements separately from each other, although in practice, multiple approaches could potentially be used in combination with each other. The job roles and years of experience of the interviewed experts are listed in Table 7.1.

Id	Job Role(s)	Years of Experience
E1	Assistant software and security engineer	3
E2	Various, e.g. security analyst or incident responder	more than 15
E3	Security specialist (focus on mobile security)	10
E4	Penetration tester	5
E5	Penetration tester	3

Table 7.1: Job roles and years of experience of interviewed experts.

Figure 7.1 illustrates how the interviewed experts estimated the effectiveness of each of the possible improvements introduced in section 7.1 based on the previously introduced scale from 1 to 5. In general, the experts largely classified the presented possible improvements as effective ways to further impede reverse engineering.

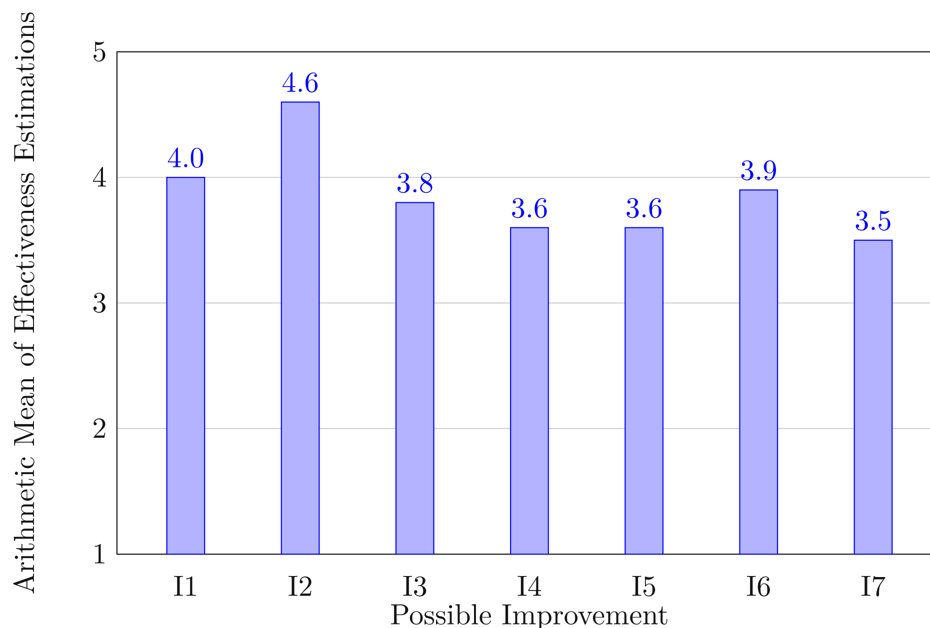


Figure 7.1: Arithmetic mean values of effectiveness estimations of presented possible improvements.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The interviewed experts considered the usage of anti-reverse engineering mechanisms in order to impede reverse engineering as reasonable and recommendable. Commonly mentioned possible consequences of reverse engineering of mobile applications include repackaging as well as theft of intellectual property. Additionally, the majority of the participating experts have already been involved in projects employing anti-reverse engineering mechanisms.

The experts classified the detection as well as prevention of the usage of debugging and hooking tools (I1 and I2, respectively) as highly effective and important. The foundation of these statements is that dynamic analysis through hooking and debugging can result in detailed insights of an application and further potentially allows to bypass various anti-reverse engineering mechanisms. Therefore, impeding the possibility of dynamic analysis represents an effective obstacle for application analysis as the detection/prevention mechanism has to be bypassed before further dynamic analysis is possible. However, the experts expressed their doubts regarding the technical feasibility of implementing a reliable mechanism that prevents the usage of debugging and hooking tools (I2), even though the experts classified preventing the usage of such tools as more effective than solely detecting them.

The interviewed experts considered running environment checks repeatedly (I3) as an important addition in order to recognise potential debugging or hooking tools that are attached later during an application's runtime. E3 noted that the repeated execution of check routines might cause performance issues and could negatively impact an application's usability. E1 and E5 advised that different entry points, e.g. different functions, should be used when distributing the calls of environment check routines across an applications' code base. Otherwise, overwriting a single function that is responsible for executing the check routines might be sufficient to bypass every further invocation. Similarly, overwriting check routines on a lower level, e.g. through modifying the return values of used C standard library functions, might lead to the bypass of check routines. E2 suggested that the locations/points in time where check routines are called during an application's runtime could be randomised on each application build. With this addition, reverse engineers would be potentially required to repeatedly perform analysis in order to bypass anti-reverse engineering mechanisms when dealing with different application builds.

The experts considered avoiding the usage of C standard library functions and calling system calls directly (I4) as an impediment to reverse engineering. However, they stated that tracing and overwriting system calls can still be done in reasonable time. Thus, the interviewed experts classified this approach as less effective compared to most of the other presented ones.

E1 and E2 saw great potential in introducing indeterminism to environment check routines (I5). Their reasoning was that indeterminism potentially requires repeated and redundant analysis of the same application, thus increasing the needed effort for reverse engineering the given application. In contrast, E3, E4, and E5 stated that, although indeterminism might lead to potentially requiring repeated and redundant analysis, in most cases the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

execution order of different environment checks or instructions does not greatly impede analysis or bypass approaches. Nonetheless, E4 noted that when randomising the reaction to a positive check result, e.g. terminating or freezing the application, the connection between the execution of the check and the corresponding reaction might not be clearly visible.

The experts considered verifying the execution of environment check routines (I6) as highly effective. All interviewed experts particularly recommended embedding the execution of check routines into mandatory initialisation procedures. By pursuing this approach, a more detailed and fine-grained analysis could be required in order to be able to specifically overwrite the check routines without affecting initialisation routines necessary for the proper execution of an application. Additionally, E2 recommended employing obfuscation techniques in order to impede a more detailed analysis, causing this approach to be more effective. However, E2 and E3 noted that a reliable implementation of this approach might be challenging. Alternatively, verifying values of certain variables that suggest the complete execution of check routines is also an effective possibility, as the respective variables have to be identified first, according to E4 and E5.

Compared to the other presented possible improvements, the experts assessed implementing advanced obfuscation techniques (I7) as the least effective approach. E4 considers the usage of advanced obfuscation techniques as an impediment for static analysis but argues that dynamic analysis can still be conducted as it is typically largely unaffected from the employment of obfuscation techniques. Most of the interviewed experts noted that also for obfuscation techniques that have just emerged or are employed less often in practise, deobfuscation or simplification tools can quickly break such techniques. In contrast, E2 and E5 are not concerned about the development of potential deobfuscation or simplification tools, as mobile application protection is an arms race between attackers and defenders, and therefore requires developing and employing new anti-reverse engineering approaches consistently.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 8

Conclusion and Future Work

Mobile anti-reverse engineering mechanisms, which are often employed through anti-reverse engineering tools, aim to impede the reverse engineering process of mobile applications. Thus, by applying anti-reversing tools and mechanisms, such as string/class encryption and various environment checks, the functionality and implementation details of applications can be concealed. Whilst developers might leverage such mechanisms to protect intellectual property and prevent misuse of their benign applications, malware developers can take advantage of anti-reversing tools and mechanisms to conceal their malicious mobile applications. Therefore, this thesis aimed to analyse Android anti-reversing mechanisms provided by commercial anti-reverse engineering tools in order to gain insights into the implementations of the analysed tools and mechanisms.

After providing fundamental information about the Android operating system, common reverse engineering techniques as well as anti-reverse engineering mechanisms, reverse engineering tools, and the analysed anti-reversing tools, we conducted practical analysis. Our analysis process built upon an evaluation application, where we applied string as well as class encryption, TLS certificate pinning, and root detection mechanisms provided by the tools *DexProtector* [145], *LIAPP* [165], and *DashO* [205], one mechanism after the other. Through static and dynamic analysis and using common reversing and diagnostic tools, such as *jadx* [229], *Frida* [95], and *strace* [160], we gained insights into the implementations of the applied anti-reversing mechanisms. Based on our analysis, we further worked out main implementation differences between the analysed tools and mechanisms. For example, analysis showed that native implementations of anti-reverse engineering mechanisms are typically more robust compared to bytecode implementations, as decompilation is impeded. In addition, techniques for preventing typical bypassing strategies through overwriting check routines can be employed by embedding checks into the initialisation routines of native libraries. Further, we observed different prioritisation approaches regarding hooking detection and distribution of entry points to environment checks.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Afterwards, we developed approaches for bypassing the analysed anti-reversing mechanisms, primarily in the form of Frida scripts. Solely in the case of LIAPP [165], where the hooking detection mechanism is integrated into the loading routine of its native library, we refrained from using Frida and made use of an Android emulator with changed system properties for bypassing LIAPP's root detection mechanism. Besides LIAPP's string and class encryption mechanisms, we were able to develop working bypassing strategies for all analysed mechanisms. Additionally, we developed a static and dynamic approach using *APKiD* [217] and *Frida* [95], respectively, for automatically determining which of the three analysed anti-reversing tools has been applied to a given application in order to be able to execute the correct bypassing scripts. Both approaches can be extended in the interest of adding support for additional anti-reversing tools.

Derived from our analysis results and bypassing strategies, we further presented possible approaches for improving the analysed anti-reversing mechanisms and tools. For example, as a majority of our analysis and bypassing procedure leveraged function hooking using Frida, we suggest requiring mandatory hooking and debugging routines to be executed. In case of a positive check result, the application could be terminated immediately before potential encrypted strings or classes are decrypted in order to prevent further dynamic analysis. Ideally, this and other mechanisms are implemented as part of a native library and avoid using functions contained in the C standard library (libc) [155] in order to impede traditional function hooking. In addition, anti-reversing tools could aim to ensure that the required/configured mechanisms have been executed properly, e.g. through embedding and verifying flags indicating a complete execution and hiding them by applying advanced obfuscation techniques, such as *Mixed Boolean-Arithmetics* [267]. We evaluated the presented possible improvements by interviewing several experts in the field of IT security. The interviewed experts mostly supported the presented ideas and assessed them as effective ways to impede reverse engineering of Android applications.

In conclusion, this work provides detailed information about the internals of anti-reverse engineering mechanisms provided by anti-reversing tools and shows how applied mechanisms can be bypassed automatically in order to enable efficient mobile application analysis. Furthermore, this work shows that the field of mobile application security and Android anti-reversing mechanisms and tools still represents an arms race between attackers and defenders and thus requires continuous research.

On the basis of the results of this thesis, future research can be conducted in two main directions. On the one hand, future works could extend the analysis of common anti-reversing mechanisms and tools, include additional tools and mechanisms, and work out further bypassing strategies in order to assist malware analysts. Since this work focused on anti-reversing mechanisms for Android applications, future works could additionally include the analysis of tools and mechanisms for iOS applications. On the other hand, in favour of developers aiming to protect their benign mobile applications, future works could build upon our findings as well as possible improvements and conduct further research on the integration of advanced hardening techniques into anti-reversing tools and mechanisms.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

APPENDIX A

Appendix

A.1 Expert Interview Guide

1. Personal Information

1.1. What is your current job role?

1.2. How many years of experience do you have in your field?

[Provision of general information and background on common anti-reverse engineering mechanisms]

2. Introductory Questions

2.1. In your opinion, how important is the usage of anti-reverse engineering mechanisms for Android applications? Please justify your answer.

2.2. In your opinion, how dangerous is reverse engineering in regards to Android applications? What are potential (negative) consequences?

2.3. Have you already employed or are you planning to employ anti-reverse engineering mechanisms for Android applications?

[Explanation of rating scheme (scale 1-5)]

3. Prioritise Debugging and Hooking Detection
[Presentation of possible improvement]

3.1. In your opinion, aiming to impede bypassing anti-reversing mechanisms in order to analyse applications, how effective is prioritising hooking and debugging detection?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.2. Please justify your rating as detailed as possible.

4. Prevent Debugging and Hooking Tools From Attaching
[Presentation of possible improvement]

4.1. In your opinion, aiming to impede bypassing anti-reversing mechanisms in order to analyse applications, how effective is preventing debugging and hooking tools from attaching?

4.2. Please justify your rating as detailed as possible.

5. Run Environment Checks Repeatedly
[Presentation of possible improvement]

5.1. In your opinion, aiming to impede bypassing environment checks in order to analyse applications, how effective is running such checks repeatedly?

5.2. Please justify your rating as detailed as possible.

6. Avoid Usage of C Standard Library Functions
[Presentation of possible improvement]

6.1. In your opinion, aiming to impede bypassing anti-reversing mechanisms in order to analyse applications, how effective is avoiding using C standard library functions?

6.2. Please justify your rating as detailed as possible.

7. Introduce Indeterminism to Environment Checks
[Presentation of possible improvement]

7.1. In your opinion, aiming to impede the analysis and bypass of environment checks, how effective is introducing indeterminism to such checks?

7.2. Please justify your rating as detailed as possible.

8. Verify Execution of Environment Check Routines
[Presentation of possible improvement]

8.1. In your opinion, aiming to impede bypassing anti-reversing mechanisms in order to analyse applications, how effective is verifying the execution of environment check routines?

8.2. Please justify your rating as detailed as possible.

9. Implement Advanced Obfuscation Techniques
[Presentation of possible improvement]

9.1. In your opinion, aiming to impede the analysis of applications and potentially implemented anti-reversing mechanisms, how effective is implementing advanced obfuscation techniques?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

9.2. Please justify your rating as detailed as possible.

[Recap of introduced possible improvements with the possibility of changing the effectiveness estimations]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

3.1 Android architecture [33]. 12
3.2 Structure of an APK file. 16
3.3 Android content provider overview and storage migration [18]. 18
3.4 Interaction between Java/Kotlin-, native code, and ART, adapted from [164]. 19
3.5 Simplified illustration of IPC via Android Binder. 22
3.6 Application signing process using AAB, adapted from [37]. 24
3.7 Man-in-the-middle attack scenario [76]. 26
3.8 Typical class encryption process on Android. 33
3.9 TLS certificate pinning process, adapted from [215]. 34
3.10 HTTP communication with and without proxy server. 42

5.1 Analysis process. 50
5.2 Screenshot of evaluation application. 51
5.3 DexProtector: Extraction of native library. 55
5.4 DexProtector: Root detection process. 60
5.5 LIAPP: Contents of protected APK. 64
5.6 LIAPP: Dialogue indicating rooted device (emulator) was detected. 67

6.1 Dynamic approach for identifying applied anti-reversing tool. 83

7.1 Arithmetic mean values of effectiveness estimations of presented possible improvements. 90

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1 Feature comparison of DexProtector, LIAPP, and DashO. 45

7.1 Job roles and years of experience of interviewed experts. 90

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Listings

3.1 Identifier renaming example. 30
3.2 Control flow obfuscation example: original code. 31
3.3 Control flow obfuscation example: obfuscated code. 31
3.4 Reflection example: original code. 31
3.5 Reflection example: obfuscated code. 31
3.6 String encryption example: original code. 32
3.7 String encryption example: code with encrypted strings. 32
3.8 TLS certificate pinning using Android's network security configuration [30]. 34

5.1 DexProtector: Example configuration file. 52
5.2 DexProtector: Loading and deletion of shared library. 53
5.3 DexProtector: `strace` output when loading native library. 54
5.4 DexProtector: Integrity check. 55
5.5 DexProtector: Replacement of `url.openConnection()` call to enforce TLS certificate pinning. 58
5.6 DexProtector: Generated class to enforce TLS certificate pinning. . . . 59
5.7 DexProtector: Root detection result in JSON. 61
5.8 DexProtector: `strace` output when checking root-indicating files. . . 62
5.9 DashO: Example configuration file. 67
5.10 DashO: Replaced strings. 68
5.11 DashO: Inserted anonymous class used for decrypting strings. 69
5.12 DashO: Root detection configuration. 70
5.13 DashO: Snippet of root detection implementation. 71

6.1 Python implementation of DashO's string decryption routine. 80

7.1 MBA transformation example [163]. 89

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

AAB Android App Bundle 23, 46, 47

AAR Android App Archive 46

AES Advanced Encryption Standard 32

AID Android ID 20

AOT Ahead-Of-Time 14

API Application Programming Interface 8, 14, 31–33, 35, 40, 43, 51, 57, 59, 65, 75, 78, 87

APK Android Application Package 15, 19, 23, 32, 36, 38, 40, 46, 47, 52, 57, 59, 60, 62–69, 71–73, 78, 81

ART Android Runtime 1, 13–15, 19, 28, 53

CA Certificate Authority 26, 27, 34, 42, 43, 57

DAC Discretionary Access Control 20, 23, 24

DEX Dalvik Executable 13, 25, 33, 38, 39, 57, 59, 63–65, 76–78, 81

DVM Dalvik Virtual Machine 13–15, 28

GID Group ID 20, 21

GPS Global Positioning System 37

HAL Hardware Abstraction Layer 11

HTTP Hypertext Transfer Protocol 26, 42, 43, 59

HTTPS Hypertext Transfer Protocol Secure 3, 26, 33, 42, 43, 51, 57

ID Identification 20, 22, 37

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

IDE Integrated Development Environment 41

IP Internet Protocol 57

IPC Inter-Process Communication 22

IT Information Technology 4, 89, 94

JAR Java Archive 38, 40

JDWP Java Debug Wire Protocol 28, 37

JIT Just-In-Time 13, 14

JNI Java Native Interface 19, 46, 54, 56, 59, 75

JSON JavaScript Object Notation 61–63, 78

JVM Java Virtual Machine 13, 28, 46

MAC Mandatory Access Control 24

MBA Mixed Boolean-Arithmetic 89

MITM Man-In-The-Middle 25–27, 33

NDK Native Development Kit 18, 19

NSA National Security Agency 39

OEM Original Equipment Manufacturer 21, 62

OS Operating System 9, 13, 15, 20, 21, 62, 64, 66, 70, 79

OTA Over-The-Air 70

OWASP Open Worldwide Application Security Project 2

PE Portable Executable 81

PID Process ID 22

RASP Runtime Application Self-Protection 6

SHA Secure Hash Algorithm 54

SMS Short Message Service 18

SSL Secure Sockets Layer 26, 42, 43

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TCP Transmission Control Protocol 40, 42

TLS Transport Layer Security xi, xiii, 1, 3, 7, 8, 26, 27, 33, 34, 39, 42, 43, 45, 46, 57–59, 73, 75, 77, 78, 93, 103

UI User Interface 17, 51

UID User ID 20, 21, 23

URL Uniform Resource Locator 17, 59

VM Virtual Machine 5, 13

WBC White-Box Cryptography 32

XML Extensible Markup Language 15, 38, 46, 47, 52, 56, 57, 66

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[51] Genevieve Arboit. "A Method for Watermarking Java Programs via Opaque Predicates". In: *The Fifth International Conference on Electronic Commerce Research (ICECR-5)* (2002).

[52] Yauhen Leanidavich Arnatovich et al. "A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation". In: *IEEE Access* 6 (2018). DOI: 10.1109/ACCESS.2018.2808340.

[53] Nitay Artenstein and Idan Revivo. "Man in the Binder: He Who Controls IPC, Controls the Droid". In: *Eur. BlackHat Conf.* 2014.

[54] Alessandro Bacci et al. "Detection of Obfuscation Techniques in Android Applications". In: *ACM International Conference Proceeding Series*. 2018. DOI: 10.1145/3230833.3232823.

[55] Vivek Balachandran et al. "Control flow obfuscation for Android applications". In: *Computers and Security* 61 (2016). DOI: 10.1016/j.cose.2016.05.003.

[56] David Barrera et al. "Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android". In: *Proceedings of the ACM Conference on Computer and Communications Security*. SPSM '12. New York, NY, USA: Association for Computing Machinery, 2012. DOI: 10.1145/2381934.2381949.

[57] Stefano Berlato and Mariano Ceccato. "A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps". In: *Journal of Information Security and Applications* 52 (2020). DOI: 10.1016/j.jisa.2020.102463.

[58] Mahesh Bhor and Deepak Karia. "Certificate Pinning for Android Applications". In: *Proceedings of the International Conference on Inventive Systems and Control, ICISC*. 2017. DOI: 10.1109/ICISC.2017.8068748.

[59] Marie Rose Boueiz. "Importance of Rooting in an Android Data Acquisition". In: *8th International Symposium on Digital Forensics and Security, ISDFS*. 2020. DOI: 10.1109/ISDFS49300.2020.9116445.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[60] Damjan Buhov et al. "Network Security Challenges in Android Applications". In: *Proceedings - 10th International Conference on Availability, Reliability and Security, ARES*. 2015. DOI: 10.1109/ARES.2015.59.

[61] Damjan Buhov et al. "Pin it! Improving Android Network Security At Runtime". In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops, IFIP Networking*. 2016. DOI: 10.1109/IFIPNetworking.2016.7497238.

[62] Luca Casati and Andrea Visconti. "The Dangers of Rooting: Data Leakage Detection in Android Applications". In: *Mobile Information Systems* (2018). DOI: 10.1155/2018/6020461.

[63] Dominic Chell et al. *The Mobile Application Hacker's Handbook*. 1st ed. Wiley, 2015. ISBN: 1118958527.

[64] Haehyun Cho, Jeong Hyun Yi, and Gail Joon Ahn. "DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications". In: *IEEE Access* 6 (2018). DOI: 10.1109/ACCESS.2018.2881699.

[65] Haehyun Cho et al. "Anti-debugging scheme for protecting mobile apps on android platform". In: *Journal of Supercomputing* 72.1 (2016). DOI: 10.1007/s11227-015-1559-9.

[66] Haehyun Cho et al. "Mobile application tamper detection scheme using dynamic code injection against repackaging attacks". In: *Journal of Supercomputing* 72.9 (2016). DOI: 10.1007/s11227-016-1763-2.

[67] Yeseul Choi et al. "EmuID: Detecting Presence of Emulation through Microarchitectural Characteristic on ARM". In: *Computers and Security* 113 (2022). DOI: 10.1016/j.cose.2021.102569.

[68] Stanley Chow et al. "White-box cryptography and an AES implementation". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2595. Springer Berlin Heidelberg, 2003. DOI: 10.1007/3-540-36492-7_17.

[69] Cristina Cifuentes and K John Gough. "Decompilation of Binary Programs". In: *Software: Practice and Experience* 25.7 (1995). DOI: 10.1002/spe.4380250706.

[70] Onur Cinar. *Android Apps with Eclipse*. 1st ed. Apress, 2012. ISBN: 1430244356.

[71] Jeremy Clark and Paul C Van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements". In: *Proceedings - IEEE Symposium on Security and Privacy*. 2013. DOI: 10.1109/SP.2013.41.

[72] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. 1997.

[73] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. New York, NY, USA: Association for Computing Machinery, 1998. DOI: 10.1145/268946.268962.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[74] Christian S. Collberg and Clark Thomborson. “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection”. In: *IEEE Transactions on Software Engineering* 28.8 (2002). DOI: 10.1109/TSE.2002.1027797.

[75] William Confer and William Roberts. *Exploring SE for Android*. Community experience distilled. Packt Publishing, 2015. ISBN: 9781784390594.

[76] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. “A Survey of Man in the Middle Attacks”. In: *IEEE Communications Surveys and Tutorials* 18.3 (2016). DOI: 10.1109/COMST.2016.2548426.

[77] Mauro Conti, P. Vinod, and Alessio Vitella. “Obfuscation detection in Android applications using deep learning”. In: *Journal of Information Security and Applications* 70 (2022). DOI: 10.1016/j.jisa.2022.103311.

[78] Jonathan Corbet, Greg Kroah-Hartman, and Alessandro Rubini. *Linux device drivers*. 3rd ed. O’Reilly & Associates, 2005. ISBN: 0596517432.

[82] Shuaike Dong et al. “Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild”. In: *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*. Vol. 254. 2018. DOI: 10.1007/978-3-030-01701-9_10.

[83] Joshua J Drake. *Android Hacker’s Handbook*. Wiley, 2014. ISBN: 9781118922255.

[84] Yue Duan et al. “Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation”. In: *NDSS*. 2018. DOI: 10.14722/ndss.2018.23296.

[85] Chris Eagle. *The IDA Pro Book*. 2nd ed. No Starch Press, 2011. ISBN: 9781593272890.

[86] Chris Eagle and Kara Nance. *The Ghidra Book*. 1st ed. No Starch Press, 2020. ISBN: 1-7185-0103-X.

[87] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android’s Security Architecture*. No Starch Press, Incorporated, 2014. ISBN: 1-59327-581-1, 978-1-59327-581-5.

[88] Michael James Van Emmerik. “Static Single Assignment for Decompilation”. PhD thesis. University of Queensland, 2007.

[89] William Enck, Machigar Ongtang, and Patrick McDaniel. “Understanding Android Security”. In: *IEEE Security and Privacy* 7.1 (2009). DOI: 10.1109/MSP.2009.26.

[90] Sascha Fahl et al. “Rethinking SSL Development in an Appified World”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2013. DOI: 10.1145/2508859.2516655.

[91] Sascha Fahl et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2012. DOI: 10.1145/2382196.2382205.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[93] Parvez Faruki et al. "Android Security: A Survey of Issues, Malware Penetration, and Defenses". In: *IEEE Communications Surveys and Tutorials* 17 (2015). DOI: 10.1109/COMST.2014.2386139.

[99] Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh. "Software Protection through Anti-Debugging". In: *IEEE Security and Privacy* 5.3 (2007). DOI: 10.1109/MSP.2007.71.

[100] D Geethanjali et al. "AEON: Android Encryption based Obfuscation". In: *CODASPY 2018 - Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*. 2018. DOI: 10.1145/3176258.3176943.

[106] Leonid Glanz et al. "CodeMatch: Obfuscation Won't Conceal Your Repackaged App". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017. DOI: 10.1145/3106237.3106305.

[107] Leonid Glanz et al. "Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS*. ACM, 2020. DOI: 10.1145/3320269.3384745.

[114] Pierre Graux. "Challenges of native android applications: obfuscation and vulnerabilities". PhD thesis. Université Rennes 1, 2020.

[115] Pierre Graux, Jean Francois Lalande, and Valérie Viet Triem Tong. "Obfuscated Android Application Development". In: *ACM International Conference Proceeding Series* (2019). DOI: 10.1145/3360664.3361144.

[116] Dawn Griffiths and David Griffiths. *Head First Android Development, 3rd edition*. O'Reilly Media, Inc., 2021. ISBN: 149207649X.

[118] Sheran Gunasekera. *Android Apps Security: Mitigate Hacking Attacks and Security Breaches*. 2nd ed. 20. Apress L. P, 2020. DOI: 10.1007/978-1-4842-1682-8.

[119] Fanglu Guo, Peter Ferrie, and Tzi Cker Chiueh. "A Study of the Packer Problem and Its Solutions". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Ed. by Richard Lippmann, Engin Kirda, and Ari Trachtenberg. Vol. 5230 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-87403-4_6.

[120] Runsheng Guo et al. "A Survey of Obfuscation and Deobfuscation Techniques in Android Code Protection". In: *Proceedings - 2022 7th IEEE International Conference on Data Science in Cyberspace, DSC*. 2022. DOI: 10.1109/DSC55868.2022.00013.

[121] James Hamilton and Sebastian Danicic. "An Evaluation of Current Java Bytecode Decompilers". In: *9th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*. 2009. DOI: 10.1109/SCAM.2009.24.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[122] Nicolas Harrand et al. "The Strengths and Behavioral Quirks of Java Bytecode Decompilers". In: *Proceedings - 19th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*. 2019. DOI: 10.1109/SCAM.2019.00019.

[123] Mohamed Hassan and Lutta Pantaleon. "An Investigation into the Impact of Rooting Android Device on User Data Integrity". In: *Proceedings - 2017 7th International Conference on Emerging Security Technologies, EST*. 2017. DOI: 10.1109/EST.2017.8090395.

[124] Vincent Haupert et al. "Honey, I Shrunk Your App Security: The State of Android App Hardening". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA*. 2018. DOI: 10.1007/978-3-319-93411-2_4.

[125] Daojing He, Sammy Chan, and Mohsen Guizani. "Mobile Application Security: Malware Threats and Defenses". In: *IEEE Wireless Communications* 22.1 (2015). DOI: 10.1109/MWC.2015.7054729.

[128] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. "SafetyNOT: On the Usage of the SafetyNet Attestation API in Android". In: *MobiSys 2021 - Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2021. DOI: 10.1145/3458864.3466627.

[129] Stefan Ilić and Slavica Dukić. "Protection of Android Applications from Decompilation Using Class Encryption and Native Code". In: *2016 Zooming Innovation in Consumer Electronics International Conference, ZINC*. 2016. DOI: 10.1109/ZINC.2016.7513642.

[130] Anurag Kumar Jain and Devendra Shanbhag. "Addressing Security and Privacy Risks in Mobile Applications". In: *IT Professional* 14.5 (2012). DOI: 10.1109/MITP.2012.72.

[132] Shuai Jiang et al. "Function-level obfuscation detection method based on Graph Convolutional Networks". In: *Journal of Information Security and Applications* 61 (2021). DOI: 10.1016/j.jisa.2021.102953.

[133] Yiming Jing et al. "Morpheus: Automatically Generating Heuristics to Detect Android Emulators". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014. DOI: 10.1145/2664243.2664250.

[135] Jin Hyuk Jung et al. "Repackaging attack on android banking applications and its countermeasures". In: *Wireless Personal Communications* 73.4 (2013). DOI: 10.1007/s11277-013-1258-x.

[136] S. Karthick and Sumitra Binu. "Android Security Issues and Solutions". In: *IEEE International Conference on Innovative Mechanisms for Industry Applications, ICIMIA 2017 - Proceedings*. 2017. DOI: 10.1109/ICIMIA.2017.7975551.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

[138] Srinivasa Rao Kotipalli and Mohammed A Imran. *Hacking Android : explore every nook and cranny of the Android OS to modify your device and guard it against security threats*. 1st ed. Community experience distilled. Packt Publishing, 2016. ISBN: 1785888005.

[139] Aleksandrina Kovacheva. "Efficient Code Obfuscation for Android". In: *International Conference on Advances in Information Technology*. Springer. 2013. DOI: 10.1007/978-3-319-03783-7_10.

[140] Rishikesh Kumar. "Android App Size Reduction: Analysis and different methodology". In: *2021 4th International Conference on Electrical, Computer and Communication Technologies, ICECCT*. 2021. DOI: 10.1109/ICECCT52121.2021.9616819.

[141] Timea László and Ákos Kiss. "Obfuscating C++ Programs via Control Flow Flattening". In: *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30.1 (2009).

[142] Sangchul Lee and Jae Wook Jeon. "Evaluating performance of Android platform using native C for embedded systems". In: *ICCAS 2010 - International Conference on Control, Automation and Systems*. 2010. DOI: 10.1109/iccas.2010.5669738.

[143] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. "Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark". In: *IEEE Transactions on Software Engineering* 47.4 (2021). DOI: 10.1109/TSE.2019.2901679.

[150] Jongsu Lim and Jeong Hyun Yi. "Structural analysis of packing schemes for extracting hidden codes in mobile malware". In: *Eurasip Journal on Wireless Communications and Networking* 2016.1 (2016). DOI: 10.1186/s13638-016-0720-3.

[151] Jie Lin, Chuanyi Liu, and Binxing Fang. "Out-of-Domain Characteristic Based Hierarchical Emulator Detection for Mobile". In: *Proceedings of the 2nd International Conference on Information Technologies and Electrical Engineering*. 2020. DOI: 10.1145/3386415.3387091.

[163] Binbin Liu et al. "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation". In: *Proceedings of the 30th USENIX Security Symposium*. 2021. ISBN: 9781939133243.

[164] Feipeng Liu. *Android Native Development Kit Cookbook*. 1st ed. Packt Publishing, 2013. ISBN: 9781849691505.

[168] Ari Luotonen and Kevin Altis. "World-Wide Web Proxies". In: *Computer Networks and ISDN Systems* 27.2 (1994). DOI: 10.1016/0169-7552(94)90128-7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[169] Dominik Maier, Tilo Muller, and Mykola Protsenko. “Divide-and-Conquer: Why Android Malware Cannot Be Stopped”. In: *Proceedings - 9th International Conference on Availability, Reliability and Security, ARES*. 2014. DOI: 10.1109/ARES.2014.12.

[170] Davide Maiorca et al. “Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware”. In: *Computers and Security* 51 (2015). DOI: 10.1016/j.cose.2015.02.007.

[171] Keith Makan and Scott Alexander-Bown. *Android Security Cookbook*. 1st ed. Packt Publishing, 2013. ISBN: 178216717X.

[172] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. “A Large-Scale Empirical Study of Android App Decompilation”. In: *Proceedings - 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*. 2021. DOI: 10.1109/SANER50967.2021.00044.

[173] René Mayrhofer et al. “The Android Platform Security Model”. In: *ACM Transactions on Privacy and Security* 24.3 (2021). DOI: 10.1145/3448609.

[174] Bill McCarty. *SELinux*. O’Reilly Media, Inc, 2004. ISBN: 9780596007164.

[175] G. Blake Meike. *Android Concurrency*. The Android deep dive series. Addison-Wesley Professional, 2016. ISBN: 9780134177618.

[176] Huasong Meng et al. “A Survey of Android Exploits in the Wild”. In: *Computers and Security* 76 (2018). DOI: 10.1016/j.cose.2018.02.019.

[177] Alessio Merlo et al. “ARMAND: Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection”. In: *Pervasive and Mobile Computing* 76 (2021). DOI: 10.1016/j.pmcj.2021.101443.

[178] Alessio Merlo et al. “You Shall not Repackage! Demystifying Anti-Repackaging on Android”. In: *Computers and Security* 103 (2021). DOI: 10.1016/j.cose.2021.102181.

[179] Georg Merzdovnik. “Security and Privacy in Mobile Environments”. PhD thesis. Technical University of Vienna, 2017.

[180] P. D. Meshram and R. C. Thool. “A survey paper on vulnerabilities in Android OS and Security of Android Devices”. In: *Proceedings - 2014 IEEE Global Conference on Wireless Computing and Networking, GCWCN 2014*. 2014. ISBN: 9781479962983. DOI: 10.1109/GCWCN.2014.7030873.

[182] Omid Mirzaei et al. “AndrODet: An adaptive Android obfuscation detector”. In: *Future Generation Computer Systems* 90 (2019). DOI: 10.1016/j.future.2018.07.066.

[183] Anmol Misra and Abhishek Dubey. *Android Security: Attacks and Defenses*. Auerbach Publications, 2013. ISBN: 9781439896464.

[185] Ginger Myles and Christian Collberg. “Software watermarking via opaque predicates: Implementation, analysis, and attacks”. In: *Electronic Commerce Research* 6.2 (2006). DOI: 10.1007/s10660-006-6955-z.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[187] Christopher Negus. *Linux Bible*. John Wiley & Sons, Incorporated, 2020. ISBN: 9781119578888.

[188] Long Nguyen-Vu et al. “Android Rooting: An Arms Race between Evasion and Detection”. In: *Security and Communication Networks* 2017 (2017). DOI: 10.1155/2017/4121765.

[189] Godfrey Nolan. *Bulletproof Android: Practical Advice for Building Secure Apps*. 1st ed. Addison-Wesley Professional, 2014. ISBN: 9780133995084.

[190] Godfrey Nolan. *Decompiling Android*. Apress, 2012. ISBN: 9781430242482.

[191] Marten Oltrogge et al. “Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications”. In: *Proceedings of the 30th USENIX Security Symposium*. 2021. ISBN: 9781939133243.

[192] Lucky Onwuzurike and Emiliano De Cristofaro. “Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps”. In: *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. WiSec ’15. Association for Computing Machinery, 2015. DOI: 10.1145/2766498.2766522.

[202] Thanasis Petsas et al. “Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware”. In: *Proceedings of the 7th European Workshop on System Security*. EuroSec ’14. Association for Computing Machinery, 2014. DOI: 10.1145/2592791.2592796.

[212] Sagar Rahalkar. *A Complete Guide to Burp Suite*. 1st ed. Apress, 2021. ISBN: 1484264029.

[213] Nick Rahimi, John Nolen, and Bidyut Gupta. “Android Security and Its Rooting — A Possible Improvement of Its Security Architecture”. In: *Journal of Information Security* 10.02 (2019). DOI: 10.4236/jis.2019.102005.

[214] Vinodha Ramasamy and Robert Hundt. “Dynamic Binary Instrumentation on IA-64”. In: *Proceedings of the First EPIC Workshop*. 2001.

[215] Francisco José Ramírez-López et al. “A Framework to Secure the Development and Auditing of SSL Pinning in Mobile Applications: The Case of Android Devices”. In: *Entropy* 21.12 (2019). DOI: 10.3390/e21121136.

[216] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. “Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks”. In: *IEEE Transactions on Information Forensics and Security* 9.1 (2014). DOI: 10.1109/TIFS.2013.2290431.

[218] Chuangang Ren, Kai Chen, and Peng Liu. “Droidmarking: Resilient Software Watermarking for Impeding Android Application Repackaging”. In: *ASE 2014 - Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 2014. DOI: 10.1145/2642937.2642977.

[219] Eric Rescorla. *HTTP Over TLS*. RFC 2818. 2000. DOI: 10.17487/RFC2818.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[220] Roman Rohleder. "Hands-On Ghidra - A Tutorial about the Software Reverse Engineering Framework". In: *SPRO 2019 - Proceedings of the 3rd ACM Workshop on Software Protection*. 2019. DOI: 10.1145/3338503.3357725.

[222] Sebastian Schrittwieser et al. "Protecting Software through Obfuscation: Can it Keep Pace with Progress in Code Analysis?" In: *ACM Computing Surveys* 49.1 (2016). DOI: 10.1145/2886012.

[225] Asaf Shabtai et al. "Google Android: A Comprehensive Security Assessment". In: *IEEE Security and Privacy* 8.2 (2010). DOI: 10.1109/MSP.2010.2.

[226] Felipe Sierra and Anthony Ramirez. "Defending Your Android App". In: *RIIT 2015 - Proceedings of the 4th Annual ACM Conference on Research in Information Technology*. 2015. DOI: 10.1145/2808062.2808067.

[227] Vikas Sihag, Manu Vardhan, and Pradeep Singh. "A Survey of Android Application and Malware Hardening". In: *Computer Science Review* 39 (2021). DOI: 10.1016/j.cosrev.2021.100365.

[228] Jeff Six. *Application Security for the Android Platform*. 1. release. O'Reilly, 2012. ISBN: 9781449315078.

[230] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android". In: *Ndss* 310 (2013).

[231] Benfano Soewito and Agung Suwandaru. "Android Sensitive Data Leakage Prevention with Rooting Detection using Java Function Hooking". In: *Journal of King Saud University - Computer and Information Sciences* (2020). DOI: 10.1016/j.jksuci.2020.07.006.

[232] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. 10th ed. Free Software Foundation, 2022. ISBN: 9780983159230.

[234] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. "Android Rooting: Methods, Detection, and Evasion". In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. 2015. DOI: 10.1145/2808117.2808126.

[236] Martin Sysel and Ondřej Doležal. "An Educational HTTP Proxy Server". In: *Procedia Engineering* 69 (2014). DOI: 10.1016/j.proeng.2014.02.212.

[237] Michał Szczepanik, Michał Kędziora, and Ireneusz Jóźwiak. "Android Methods Hooking Detection Using Dalvik Code and Dynamic Reverse Engineering by Stack Trace Analysis". In: *Advances in Intelligent Systems and Computing*. Vol. 1173. 2020. DOI: 10.1007/978-3-030-48256-5_62.

[242] Nikolaos Totosis and Constantinos Patsakis. "Android Hooking Revisited". In: *IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. 2018. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00104.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[244] Jason Tyler. *XDA Developers' Android Hacker's Toolkit: The Complete Guide to Rooting, ROMs and Theming*. 2nd ed. John Wiley & Sons, 2012. ISBN: 1119961556.

[246] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. "Deobfuscation Reverse Engineering Obfuscated Code". In: *12th Working Conference on Reverse Engineering (WCRE'05)*. Vol. 2005. IEEE, 2005. ISBN: 0769524745. DOI: 10.1109/WCRE.2005.13.

[247] Timothy Vidas and Nicolas Christin. "Evading Android Runtime Analysis via Sandbox Detection". In: *ASIA CCS 2014 - Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security* (2014). DOI: 10.1145/2590296.2590325.

[248] Timothy Vidas, Daniel Votipka, and Nicolas Christin. "All Your Droid Are Belong To Us: A Survey of Current Android Attacks". In: *5th USENIX Workshop on Offensive Technologies, WOOT*. 2011. DOI: 10.5555/2028052.2028062.

[249] Timothy Vidas, Chengye Zhang, and Nicolas Christin. "Toward a General Collection Methodology for Android Devices". In: *Digital Investigation* 8.SUPPL. (2011). DOI: 10.1016/j.diin.2011.05.003.

[250] Lori Vinciguerra et al. "An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java". In: *Proceedings - Working Conference on Reverse Engineering, WCRE*. 2003. DOI: 10.1109/WCRE.2003.1287233.

[253] Jia Wan, Mohammad Zulkernine, and Clifford Liem. "A Dynamic App Anti-Debugging Approach on Android ART Runtime". In: *IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. 2018. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00105.

[254] Haoyu Wang et al. "Characterizing Android App Signing Issues". In: *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. 2019. DOI: 10.1109/ASE.2019.00035.

[255] Le Wang and Alexander M. Wyglinski. "Detection of Man-in-the-Middle Attacks Using Physical Layer Wireless Security Techniques". In: *Wireless Communications and Mobile Computing* 16.4 (2016). DOI: 10.1002/wcm.2527.

[256] Xuetao Wei and Michael Wolf. "A Survey on HTTPS Implementation by Android Apps: Issues and Countermeasures". In: *Applied Computing and Informatics* 13.2 (2017). DOI: 10.1016/j.aci.2016.10.001.

[257] Dominik Wermke et al. "A Large Scale Investigation of Obfuscation Use in Google Play". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018. DOI: 10.1145/3274694.3274726.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[259] Dongpeng Xu, Jiang Ming, and Dinghao Wu. "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method". In: *Information Security*. Springer International Publishing, 2016. ISBN: 978-3-319-45871-7.

[260] Meng Xu et al. "Toward Engineering a Secure Android Ecosystem: A Survey of Existing Techniques". In: *ACM Computing Surveys* 49.2 (2016). DOI: 10.1145/2963145.

[261] Woojong Yoo et al. "String Deobfuscation Scheme based on Dynamic Code Extraction for Mobile Malwares". In: *IT Convergence Practice* 4.2 (2016).

[262] Xian Zhan, Tao Zhang, and Yutian Tang. "A Comparative Study of Android Repackaged Apps Detection Techniques". In: *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*. 2019. DOI: 10.1109/SANER.2019.8667975.

[263] Hang Zhang, Dongdong She, and Zhiyun Qian. "Android Root and Its Providers: A Double-Edged Sword". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015. DOI: 10.1145/2810103.2813714.

[264] Mu Zhang and Heng Yin. *Android Application Security*. SpringerBriefs in Computer Science. Springer International Publishing, 2016. ISBN: 978-3-319-47811-1.

[265] Xiaolu Zhang et al. "Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations". In: *Forensic Science International: Digital Investigation* 39 (2021). DOI: 10.1016/j.fsidi.2021.301285.

[266] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. "DexHunter: Toward Extracting Hidden Code from Packed Android Applications". In: *Lecture Notes in Computer Science*. Vol. 9327. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-24177-7_15.

[267] Yongxin Zhou et al. "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms". In: *Information Security Applications*. Springer Berlin Heidelberg, 2007. ISBN: 978-3-540-77535-5.

[268] Lukas Zobernig, Steven D Galbraith, and Giovanni Russello. "When are Opaque Predicates Useful?" In: *Proceedings - 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering, TrustCom/BigDataSE*. 2019. DOI: 10.1109/TrustCom/BigDataSE.2019.00031.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Online References

[1] Aldeid. *PEiD*. 2020. URL: https://www.aldeid.com/wiki/PEiD (visited on 08/19/2023).

[2] *Android*. 2023. URL: https://www.android.com/ (visited on 08/19/2023).

[3] Android Code Search. *ClassLinker*. 2023. URL: https://cs.android.com/android/platform/superproject/+/master:art/runtime/class_linker.cc;l=3228;drc=master (visited on 08/19/2023).

[4] Android Code Search. *DexFileLoader*. 2023. URL: https://cs.android.com/android/platform/superproject/+/master:art/libdexfile/dex/dex_file_loader.cc;l=341 (visited on 08/19/2023).

[5] Android Developers. *About Android App Bundles*. 2023. URL: https://developer.android.com/guide/app-bundle (visited on 08/19/2023).

[6] Android Developers. *ActivityManager*. 2023. URL: https://developer.android.com/reference/android/app/ActivityManager (visited on 08/19/2023).

[7] Android Developers. *Android Debug Bridge (adb)*. 2023. URL: https://developer.android.com/studio/command-line/adb (visited on 08/19/2023).

[8] Android Developers. *Android NDK*. 2023. URL: https://developer.android.com/ndk (visited on 08/19/2023).

[9] Android Developers. *Android Studio*. 2023. URL: https://developer.android.com/studio (visited on 08/19/2023).

[10] Android Developers. *App Manifest Overview*. 2023. URL: https://developer.android.com/guide/topics/manifest/manifest-intro (visited on 08/19/2023).

[11] Android Developers. *AppComponentFactory*. 2023. URL: https://developer.android.com/reference/android/app/AppComponentFactory (visited on 08/19/2023).

[12] Android Developers. *Application*. 2023. URL: https://developer.android.com/reference/android/app/Application (visited on 08/19/2023).

[13] Android Developers. *Application Fundamentals*. 2023. URL: https://developer.android.com/guide/components/fundamentals (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

[14] Android Developers. *Bound services overview*. 2023. URL: https://developer.android.com/guide/components/bound-services (visited on 08/19/2023).

[15] Android Developers. *Build*. 2023. URL: https://developer.android.com/reference/android/os/Build (visited on 08/19/2023).

[16] Android Developers. *CertificateException*. 2023. URL: https://developer.android.com/reference/java/security/cert/CertificateException (visited on 08/19/2023).

[17] Android Developers. *ClassLoader*. 2023. URL: https://developer.android.com/reference/java/lang/ClassLoader (visited on 08/19/2023).

[18] Android Developers. *Content Providers*. 2023. URL: https://developer.android.com/guide/topics/providers/content-providers (visited on 08/19/2023).

[19] Android Developers. *Create an Android library*. 2023. URL: https://developer.android.com/studio/projects/android-library.html (visited on 08/19/2023).

[20] Android Developers. *Debug*. 2023. URL: https://developer.android.com/reference/android/os/Debug (visited on 08/19/2023).

[21] Android Developers. *Debug your app*. 2023. URL: https://developer.android.com/studio/debug/ (visited on 08/19/2023).

[22] Android Developers. *DexClassLoader*. 2023. URL: https://developer.android.com/reference/dalvik/system/DexClassLoader (visited on 08/19/2023).

[23] Android Developers. *DexFile*. 2023. URL: https://developer.android.com/reference/dalvik/system/DexFile (visited on 08/19/2023).

[24] Android Developers. *Enable multidex for apps with over 64K methods*. 2023. URL: https://developer.android.com/build/multidex (visited on 08/19/2023).

[25] Android Developers. *File*. 2023. URL: https://developer.android.com/reference/java/io/File (visited on 08/19/2023).

[26] Android Developers. *Fragments*. 2023. URL: https://developer.android.com/guide/fragments (visited on 08/19/2023).

[27] Android Developers. *InMemoryDexClassLoader*. 2023. URL: https://developer.android.com/reference/dalvik/system/InMemoryDexClassLoader (visited on 08/19/2023).

[28] Android Developers. *Intents and Intent Filters*. 2023. URL: https://developer.android.com/guide/components/intents-filters (visited on 08/19/2023).

[29] Android Developers. *JNI tips*. 2023. URL: https://developer.android.com/training/articles/perf-jni (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[30] Android Developers. *Network security configuration*. 2022. URL: https://developer.android.com/training/articles/security-config.html (visited on 08/19/2023).

[31] Android Developers. *PackageManager*. 2023. URL: https://developer.android.com/reference/android/content/pm/PackageManager (visited on 08/19/2023).

[32] Android Developers. *Permissions on Android*. 2023. URL: https://developer.android.com/guide/topics/permissions/overview (visited on 08/19/2023).

[33] Android Developers. *Platform Architecture*. 2023. URL: https://developer.android.com/guide/platform (visited on 08/19/2023).

[34] Android Developers. *Play Integrity API*. 2023. URL: https://developer.android.com/google/play/integrity/overview (visited on 08/19/2023).

[35] Android Developers. *SafetyNet Attestation API*. 2023. URL: https://developer.android.com/training/safetynet/attestation (visited on 08/19/2023).

[36] Android Developers. *Shrink, obfuscate, and optimize your app*. 2023. URL: https://developer.android.com/studio/build/shrink-code (visited on 08/19/2023).

[37] Android Developers. *Sign your app*. 2023. URL: https://developer.android.com/studio/publish/app-signing (visited on 08/19/2023).

[38] Android Developers. *System*. 2023. URL: https://developer.android.com/reference/java/lang/System (visited on 08/19/2023).

[39] Android Developers. *The Android Profiler*. 2023. URL: https://developer.android.com/studio/profile/android-profiler (visited on 08/19/2023).

[40] Android Developers. *TrustManager*. 2019. URL: https://developer.android.com/reference/javax/net/ssl/TrustManager (visited on 08/19/2023).

[41] Android Developers. *URL*. 2023. URL: https://developer.android.com/reference/java/net/URL (visited on 08/19/2023).

[42] Android Open Source Project. *Add System Properties*. 2023. URL: https://source.android.com/docs/core/architecture/configuration/add-system-properties (visited on 08/19/2023).

[43] Android Open Source Project. *Adding a New Device*. 2023. URL: https://source.android.com/docs/setup/create/new-device (visited on 08/19/2023).

[44] Android Open Source Project. *Android Runtime (ART) and Dalvik*. 2022. URL: https://source.android.com/devices/tech/dalvik?hl=en (visited on 08/19/2023).

[45] Android Open Source Project. *Configuring ART*. 2023. URL: https://source.android.com/devices/tech/dalvik/configure (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[46] Android Open Source Project. *Dalvik bytecode*. 2022. URL: https://source.android.com/docs/core/runtime/dalvik-bytecode (visited on 08/19/2023).

[47] Android Open Source Project. *Dalvik executable format*. 2022. URL: https://source.android.com/docs/core/runtime/dex-format (visited on 08/19/2023).

[48] Android Open Source Project. *Signing Builds for Release*. 2023. URL: https://source.android.com/docs/core/ota/sign_builds (visited on 08/19/2023).

[49] Android Open Source Project. *Writing SELinux Policy*. 2022. URL: https://source.android.com/docs/security/features/selinux/device-policy#overuse_of_negation (visited on 08/19/2023).

[50] Apache. *Apache Harmony - Open Source Java Platform*. 2010. URL: https://harmony.apache.org/ (visited on 08/19/2023).

[79] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. *mitmproxy: A free and open source interactive HTTPS proxy*. 2023. URL: https://mitmproxy.org/ (visited on 08/19/2023).

[80] Darvin's Blog. *Detecting Magisk Hide*. 2019. URL: https://darvincitech.wordpress.com/2019/11/04/detecting-magisk-hide/ (visited on 08/19/2023).

[81] Darvin's Blog. *Security hardening of Android native code*. 2020. URL: https://darvincitech.wordpress.com/2020/01/07/security-hardening-of-android-native-code/ (visited on 08/19/2023).

[92] Parvez Faruki et al. *Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions*. 2016. URL: https://arxiv.org/abs/1611.10231 (visited on 08/19/2023).

[94] Free Software Foundation Inc. *GDB: The GNU Project Debugger*. 2023. URL: https://www.sourceware.org/gdb/ (visited on 08/19/2023).

[95] *Frida*. 2023. URL: https://frida.re/ (visited on 08/19/2023).

[96] Frida. *JavaScript API*. 2023. URL: https://frida.re/docs/javascript-api/ (visited on 08/19/2023).

[97] Frida. *Stalker*. 2023. URL: https://frida.re/docs/stalker/ (visited on 08/19/2023).

[98] *Frida: C API*. 2023. URL: https://frida.re/docs/c-api/ (visited on 08/19/2023).

[101] GitHub: CalebFenton. *dex-oracle*. 2018. URL: https://github.com/CalebFenton/dex-oracle (visited on 08/19/2023).

[102] GitHub: Google. *conscrypt – TrustManagerImpl.java*. 2022. URL: https://github.com/google/conscrypt/blob/master/common/src/main/java/org/conscrypt/TrustManagerImpl.java (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[103] GitHub: JesusFreke. *smali*. 2022. URL: https://github.com/JesusFreke/smali (visited on 08/19/2023).

[104] GitHub: LSPosed. *LSPosed*. 2023. URL: https://github.com/LSPosed/LSPosed (visited on 08/19/2023).

[105] GitHub: Nightbringer21. *fridump*. 2019. URL: https://github.com/Nightbringer21/fridump (visited on 08/19/2023).

[108] Sihan Goi. *Diving down the Magisk rabbit hole*. 2020. URL: https://medium.com/csg-govtech/diving-down-the-magisk-rabbit-hole-aaf88a8c2de0 (visited on 08/19/2023).

[109] Google Git. *bionic*. 2023. URL: https://android.googlesource.com/platform/bionic/ (visited on 08/19/2023).

[110] Google Git. *Rules for Isolated Apps*. 2023. URL: https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/isolated_app.te (visited on 08/19/2023).

[111] Google Git. *System properties*. 2023. URL: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/java/android/os/Build.java (visited on 08/19/2023).

[112] *Google Play Store*. 2023. URL: https://play.google.com/store (visited on 08/19/2023).

[113] Gradle Inc. *Gradle Build Tool*. 2023. URL: https://gradle.org/ (visited on 08/19/2023).

[117] Guardsquare. *ProGuard*. 2023. URL: https://www.guardsquare.com/proguard (visited on 08/19/2023).

[126] Hex-Rays. *IDA Pro*. 2023. URL: https://hex-rays.com/ida-pro/ (visited on 08/19/2023).

[127] GitHub: hluwa. *frida-dexdump*. 2022. URL: https://github.com/hluwa/FRIDA-DEXDump (visited on 08/19/2023).

[131] *Java-Decompiler*. 2023. URL: http://java-decompiler.github.io/ (visited on 08/19/2023).

[134] *JTrace – An Android Aware strace(1)*. 2023. URL: http://newandroidbook.com/tools/jtrace.html (visited on 08/19/2023).

[137] GitHub: kittinunf. *fuel*. 2023. URL: https://github.com/kittinunf/fuel (visited on 08/19/2023).

[144] Licel. *Configuring DexProtector*. 2023. URL: https://licelus.com/products/dexprotector/docs/android/configuring-dexprotector (visited on 08/19/2023).

[145] Licel. *DexProtector*. 2023. URL: https://licelus.com/products/dexprotector (visited on 08/19/2023).

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[146] Licel. *DexProtector Documentation*. 2023. URL: https://licelus.com/products/dexprotector/docs (visited on 08/19/2023).

[147] Licel. *DexProtector Feature Matrix*. 2023. URL: https://licelus.com/downloads/DexProtector_Feature_Matrix.pdf (visited on 08/19/2023).

[148] Licel. *DexProtector User Manual*. 2023. URL: https://dexprotector.com/docs (visited on 08/19/2023).

[149] *Licel*. 2023. URL: https://licelus.com/ (visited on 08/19/2023).

[152] Linux Programmer's Manual. *access(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/access.2.html (visited on 08/19/2023).

[153] Linux Programmer's Manual. *intro(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/intro.2.html (visited on 08/19/2023).

[154] Linux Programmer's Manual. *ioctl(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/ioctl.2.html (visited on 08/19/2023).

[155] Linux Programmer's Manual. *libc(7)*. 2023. URL: https://man7.org/linux/man-pages/man7/libc.7.html (visited on 08/19/2023).

[156] Linux Programmer's Manual. *open(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/open.2.html (visited on 08/19/2023).

[157] Linux Programmer's Manual. *proc(5)*. 2023. URL: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 08/19/2023).

[158] Linux Programmer's Manual. *ptrace(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 08/19/2023).

[159] Linux Programmer's Manual. *stat(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/lstat.2.html (visited on 08/19/2023).

[160] Linux Programmer's Manual. *strace(1)*. 2023. URL: https://man7.org/linux/man-pages/man1/strace.1.html (visited on 08/19/2023).

[161] Linux Programmer's Manual. *unlink(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/unlink.2.html (visited on 08/19/2023).

[162] Linux Programmer's Manual. *write(2)*. 2023. URL: https://man7.org/linux/man-pages/man2/write.2.html (visited on 08/19/2023).

[165] Lockin Company. *LIAPP*. 2023. URL: https://liapp.lockincomp.com/ (visited on 08/19/2023).

[166] Lockin Company. *LIAPP Guide*. 2023. URL: https://guide.lockincomp.com/ (visited on 08/19/2023).

[167] *Lockin Company*. 2023. URL: https://liapp.lockincomp.com/contact (visited on 08/19/2023).

[181] Microsoft. *PE Format*. 2023. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[184] *musl libc*. 2022. URL: https://musl.libc.org/ (visited on 08/19/2023).

[186] National Security Agency. *Ghidra*. 2023. URL: https://ghidra-sre.org/ (visited on 08/19/2023).

[193] Oracle. *Java Debug Wire Protocol*. 2023. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html (visited on 08/19/2023).

[194] Oracle. *JNI APIs and Developer Guides*. 2023. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/ (visited on 08/19/2023).

[195] Oracle. *JNI Functions*. 2023. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html (visited on 08/19/2023).

[196] Oracle. *The Reflection API*. 2022. URL: https://docs.oracle.com/javase/tutorial/reflect/index.html (visited on 08/19/2023).

[197] OWASP. *Mobile Application Security Verification Standard*. 2023. URL: https://mobile-security.gitbook.io/masvs/ (visited on 08/19/2023).

[198] OWASP. *Mobile Security Testing Guide*. 2023. URL: https://mobile-security.gitbook.io/mobile-security-testing-guide/ (visited on 08/19/2023).

[199] OWASP. *OWASP Mobile Top 10*. 2023. URL: https://owasp.org/www-project-mobile-top-10/ (visited on 08/19/2023).

[200] *OWASP Foundation*. 2023. URL: https://owasp.org/ (visited on 08/19/2023).

[201] PalmSource Inc. *OpenBinder*. 2005. URL: http://www.angryredplanet.com/~hackbod/openbinder/docs/html/ (visited on 08/19/2023).

[203] PortSwigger. *Burp Suite – Application Security Testing Software*. 2023. URL: https://portswigger.net/burp (visited on 08/19/2023).

[204] *PortSwigger*. 2023. URL: https://portswigger.net/ (visited on 08/19/2023).

[205] PreEmptive. *DashO*. 2023. URL: https://www.preemptive.com/products/dasho/ (visited on 08/19/2023).

[206] PreEmptive. *DashO Features*. 2023. URL: https://www.preemptive.com/products/dasho/features/ (visited on 08/19/2023).

[207] PreEmptive. *DashO User Guide*. 2023. URL: https://www.preemptive.com/dasho/pro/userguide/en/ (visited on 08/19/2023).

[208] *PreEmptive*. 2023. URL: https://www.preemptive.com/ (visited on 08/19/2023).

[209] *Promon*. 2023. URL: https://promon.co/ (visited on 08/19/2023).

[210] GitHub: pxb1988. *dex2jar*. 2021. URL: https://github.com/pxb1988/dex2jar (visited on 08/19/2023).

[211] Python Software Foundation. *Python*. 2023. URL: https://www.python.org/ (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[217] GitHub: rednaga. *APKiD*. 2023. URL: https://github.com/rednaga/APKiD (visited on 08/19/2023).

[221] Saurik. *Cydia Substrate*. 2014. URL: http://www.cydiasubstrate.com/ (visited on 08/19/2023).

[223] GitHub: scottyab. *rootbeer*. 2021. URL: https://github.com/scottyab/rootbeer (visited on 08/19/2023).

[224] GitHub: sensepost. *objection*. 2023. URL: https://github.com/sensepost/objection (visited on 08/19/2023).

[229] GitHub: skylot. *jadx*. 2023. URL: https://github.com/skylot/jadx (visited on 08/19/2023).

[233] StatCounter Global Stats. *Mobile Operating System Market Share Worldwide*. 2023. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (visited on 08/19/2023).

[235] SuperSU. *SuperSU Root*. 2020. URL: https://supersuroot.org/ (visited on 08/19/2023).

[238] *Thales*. 2023. URL: https://www.thalesgroup.com/ (visited on 08/19/2023).

[239] The LLDB Team. *The LLDB Debugger*. 2023. URL: https://lldb.llvm.org/ (visited on 08/19/2023).

[240] *The LLVM Compiler Infrastructure Project*. 2023. URL: https://llvm.org/ (visited on 08/19/2023).

[241] Topjohnwu. *Magisk Manager*. 2023. URL: https://magiskmanager.com/ (visited on 08/19/2023).

[243] Connor Tumbleson. *apktool*. 2023. URL: https://ibotpeaches.github.io/Apktool/ (visited on 08/19/2023).

[245] Typicode. *JSONPlaceholder – Free Fake REST API*. 2022. URL: https://jsonplaceholder.typicode.com/ (visited on 08/19/2023).

[251] VirusTotal. *YARA - The pattern matching swiss knife for malware researchers*. 2023. URL: https://virustotal.github.io/yara/ (visited on 08/19/2023).

[252] Jeffery Walton et al. *Certificate and Public Key Pinning Control – OWASP Foundation*. 2023. URL: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning (visited on 08/19/2023).

[258] XDA. *Xposed Framework*. 2012. URL: https://forum.xda-developers.com/t/xposed-general-info-versions-changelog.2714053/ (visited on 08/19/2023).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub