Bachelor Thesis Project

Evaluating Dynamic Analysis Methods for Android Applications

Author: Alexander Spottka
Supervisor: Ola Flygt
External Supervisor: Philip Åkesson
Semester: VT 2017
Subject: Computer Science
Abstract

With a market share of 84.82% in 2016 Android is the most influential mobile operating system on the world [1]. In March 2017 users could find about 2.8 million applications in the official Playstore while the number applications from other sources is unknown [2]. Since mobile devices are a fundamental source for news, entertainment, social activities and more they are also used for mobile banking, health tracking and other data sensitive tasks. Besides static analysis the approach of dynamically analyzing applications is necessary to ensure integrity and security. In the internet a plethora of dynamic analysis methods for Android can be found. Problematic for a software security tester is to keep an overview over the quickly changing landscape of these approaches. In this thesis work relevant dynamic analysis methods were grouped and evaluated on different criterion. Furthermore an implementation for the logging file related system calls with LD_PRELOAD was implemented and investigated how API calls can be mapped and the data visualized.
Acknowledgements

Before diving into the report I want to thank Ola Flygt for his support on this project as supervisor, as well for my time at Linnaeus university. Furthermore, I want to thank TrueSec Syd AB, and especially Emil Kvarnhammar and Philip Åkesson for showing honest and immediate interest in developing this work, supporting me the entire way. I want to thank Hamid Kashfi, who helped me taking my first steps in the area of application testing and who always had a friendly ear for me. On personal side I want to thank my girlfriend Melina, who enabled me doing my best here in Sweden, as well as my family in Germany which despite the distance always had my back.
## Contents

### List of Figures

v

### List of Tables

vi

### 1 Introduction

1.1 Background ............................................. 1
1.2 Previous Research ..................................... 1
1.3 Problem Formulation .................................. 1
1.4 Motivation ............................................ 2
1.5 Research Questions ................................. 2
1.6 Limitation ............................................ 2
1.7 Target Group ......................................... 2
1.8 Outline ............................................... 3

### 2 Method

2.1 Scientific Approach ................................. 4
2.2 Method Description ................................... 4
2.3 Reliability and Validity ............................ 5
2.4 Ethical Considerations .............................. 5

### 3 Theoretical Context

3.1 General Android Architecture ........................ 6
3.2 Android Security Architecture ...................... 7
  3.2.1 Kernel Security and Application Sandbox .......... 7
  3.2.2 Interprocess Communication (IPC) ................. 8
  3.2.3 Application Signing and Permission Model .......... 8
3.3 Development of Android Applications ................. 9
3.4 Structure of an APK ................................ 9
3.5 Definition Dynamic Analysis ........................ 9
3.6 Type-Safety ........................................... 9

### 4 Evaluation Of Dynamic Analysis Methods

4.1 Scope ............................................... 10
4.2 Metrics .............................................. 11
4.3 Dalvik / ART Debugging ............................... 11
  4.3.1 Setup ........................................... 12
  4.3.2 Experiment ....................................... 13
  4.3.3 Application for Dalvik/ART debugging ........... 15
  4.3.4 Results .......................................... 16
4.4 Native Code Debugging ............................... 17
  4.4.1 Experiment ....................................... 18
  4.4.2 GNU Debugger (GDB) ............................ 18
  4.4.3 Application for native code debugging .......... 19
  4.4.4 Results .......................................... 19
4.5 Automated Analyzers ................................. 20
  4.5.1 Tracedroid ....................................... 21
  4.5.2 NVISO ApkScan .................................. 22
  4.5.3 Visual Threat .................................... 22
# Table of Contents

4.5.4  App-Ray .................................................. 22
4.5.5  Experiment ............................................... 22
4.5.6  Application for automated web analyzers ................. 23
4.5.7  Results .................................................. 24

4.6  LD_PRELOAD ............................................... 25
4.6.1  Introduction .............................................. 25
4.6.2  Application for LD_PRELOAD ......................... 25
4.6.3  Compiling C/C++ code .................................. 26
4.6.4  Setup ..................................................... 26
4.6.5  Experiment ................................................. 27
4.6.6  Results .................................................. 29

4.7  Frida ........................................................ 30
4.7.1  The choice of analyzing Frida ............................ 30
4.7.2  Architecture ............................................. 30
4.7.3  Setup ..................................................... 31
4.7.4  Experiment ................................................. 31
4.7.5  Applications for Frida ................................. 33
4.7.6  Results .................................................. 34

5  Implementation .............................................. 36
5.1  Starting point ............................................... 36
5.2  Client Side ................................................ 36
  5.2.1  Structure of the shared library ....................... 37
5.3  Server Side ................................................ 37
5.4  Visualization .............................................. 40
5.5  Traceability of API calls .................................. 40

6  Analysis ...................................................... 42
6.1  Comparison of metrics ..................................... 44

7  Discussion ................................................... 45
7.1  Theoretical context ....................................... 45
7.2  Dynamic analysis methods ................................. 45
7.3  Traceability and Visualization ............................ 46
7.4  Application of different dynamic analysis methods ...... 46

8  Conclusion ................................................... 47
8.1  Future research ........................................... 47

References ................................................... 48
List of Figures

3.1 Android Architecture ............................................. 6
4.2 Example of an Android manifest file snippet .................. 12
4.3 Screenshot of the decompiled application BA2 .................. 14
4.4 Screenshot of the decompiled application BA3 .................. 15
4.5 Screenshot of the debugging process of InsecureBankv2 ....... 16
4.6 Failure message of Tracedroid .................................. 23
4.7 Failure message of Apkscan ..................................... 23
4.8 Sample output of the LD_PRELOAD implementation .......... 28
4.9 Architecture of FRIDA ........................................... 31
4.10 Sample output of frida-trace on the BA1 ....................... 32
4.11 Sample output of frida-trace on BA3 ........................... 33
4.12 Sample output of frida-trace ................................. 34
5.13 Components of implementation .............................. 36
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Research questions</td>
<td>2</td>
</tr>
<tr>
<td>4.2</td>
<td>Evaluation Results of Dalvik/ART debugging</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Evaluation results of native code debugging</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Evaluation results of automated web analyzers</td>
<td>25</td>
</tr>
<tr>
<td>4.5</td>
<td>Function calls of BA1</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>Function calls of BA3</td>
<td>28</td>
</tr>
<tr>
<td>4.7</td>
<td>Function calls of the InsecureBankv2 application</td>
<td>29</td>
</tr>
<tr>
<td>4.8</td>
<td>Evaluation result of the ld_preload implementation</td>
<td>30</td>
</tr>
<tr>
<td>4.9</td>
<td>Evaluation results of Frida</td>
<td>35</td>
</tr>
<tr>
<td>5.10</td>
<td>Conceptual call mapping using LD_PRELOAD</td>
<td>41</td>
</tr>
<tr>
<td>6.11</td>
<td>Comparison of evaluation results</td>
<td>42</td>
</tr>
<tr>
<td>6.12</td>
<td>Comparison of the metric scores</td>
<td>44</td>
</tr>
</tbody>
</table>
1 Introduction

The introduction section will give insight to the main concepts of the thesis. Furthermore, it will provide the reader with frame of information necessary to fully understand the outcome of this work.

1.1 Background

Every computer, smartphone, tablet, wearable or any other Internet of Things (IoT) device has software running on it. These devices are not only used for work anymore, but has taken over large parts of our lives, starting from entertainment, communication, working, banking, and health tracking. This increasing influence on private life makes securing the software running on those devices a necessity [3]. The Open Web Application Security Project (OWASP) online community, published a report from the Paladion Mobile Security Team, which stated that there is a 71% chance of any application getting misused [4]. An example for the need of security software testing is that the usage and service of online banking applications should not increase the risk of becoming the victim of an attack. If any vulnerability in such an application were to become public, the results would be loss of credibility, and fewer customers. Since securing an application is not a trivial task, expert companies like TrueSec are hired by customers to perform security tests on systems and applications. Security testing is performed to ensure that applications will follow the best practices, that communication is properly secured, and that no sensitive data is leaked. One approach to do so is static analysis, which is time-consuming and tedious. Dynamic analysis can help improving the workflow for tester. More technical information is formulated in the problem formulation.

1.2 Previous Research

In the technical report "Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation" Hoffmann et.al takes an approach of evaluating several analysis tools for the Android platform [5]. The difference is that they have a broader scope by including static analysis and focus heavily on the evasion of the analysis procedures and the effectiveness of the respective decompilers. Another conducted report was the master thesis of Victor van der Veen, in which he presents and explains the functionality of his dynamic analysis tool Tracedroid, as well as further information on the topic of dynamic analysis. In a later section of this report, Tracedroid’s functionality will be further evaluated. There are several websites and blogs that focus on the evaluation of solely static analysis tools and not dynamic analysis.

1.3 Problem Formulation

There are different dynamic analysis methods that promise exceptional test results for mobile application on the Android platform. With the high number of tools that can be found, there is a missing overview to determine which approach is applicable to specific problems. One way to analyze an application dynamically is the use of the Linux environment variable LD_PRELOAD, which will be further explained in section 4 The use of LD_PRELOAD is a highly manual process and can be improved by building a framework that is extensible and is able to present logged data in an external web application. The mapping of related calls and visualization of the data intercepted is another problem to be investigated in this project.
1.4 Motivation

The idea of this thesis was developed together with Emil Kvarnhammar (CEO TrueSec Syd AB) and Philip Åkesson (Software Security Engineer). The expected result will be a benefit for TrueSec and the security community in having current dynamic analysis methods analyzed as well as having a tool that will be developed as a proof of concept. Time is one of the most valuable resources for IT experts. By having the possibility to quickly overview dynamic analysis methods, a software security tester is able to organize his later procedures faster, save time, and therefore costs. Furthermore, the identification of related function calls and the trace analysis, including the extensible modular back-end, is going to improve the workflow of security code testers of TrueSec.

1.5 Research Questions

The research questions shown below, will be abbreviated with RQ#, were the hash tag is a placeholder for the number of the question.

| RQ1 | What dynamic analysis approaches are used on the Android platform and how can they be evaluated? |
| RQ2 | How is it possible to keep track and map API calls, such as file operations on the same file descriptor, that are related to each other? |
| RQ3 | In which scope can the methods investigated in RQ1 be applied? |
| RQ4 | How can the data extrapolated from the project be analyzed or visualized and what are the benefits for the software security tester? |

Table 1.1: Research questions

1.6 Limitation

This thesis does not intend to evaluate every dynamic analysis method that can be found, but the ones that seem suitable to employ; since the landscape of tools and methods changes frequently, the selection had to be limited. Furthermore, it is important to mention that the goal of this thesis is not to perform deep vulnerability analyses on every application with all dynamic analysis methods but rather to show the advantages and disadvantages of different approaches. Moreover, the dynamic analysis methods covered, are not evaluated on their detection rate of vulnerabilities. In addition, the results are limited to the resources found, as well as the author’s capabilities regarding dynamic analysis. Another very experienced tester might disagree with the results of this thesis or is able to add more information. The dynamic analysis methods are partially evaluated on experiments with banking applications. This focus could yield different results concerning applications of a less critical sector.

1.7 Target Group

The field that this thesis affects is specific and mainly meant for software security testers. This work might demonstrate different approaches that a tester has not used yet and might want in the future. Software security engineers should use this work to reflect on their current workflow. Moreover, mobile software developers and project managers could have a potential interest, since this thesis shows several ways of how a tester can tamper with applications in order to find vulnerabilities. Basic knowledge of the Linux and Android platform are necessary in order to fully understand the outcome of this work. Furthermore,
an understanding of how different programming languages are processed by computers is beneficial.

1.8 Outline

Chapter 2 will introduce the reader to the way this scientific work is carried out. To guide the reader into the area of Android application testing and particularly dynamic analysis, a brief introduction about the Android framework, and other necessary fundamentals is given in chapter 3. Afterwards, the implementation, including code fragments will be presented to show the interested reader a few technical details about the architecture. This will be done in chapter 5 gives answers to research question four. The main part of evaluating different dynamic analysis methods is done in chapter 4. Every method will have an encapsulated conclusion in itself. Thereafter, the conclusions of each method will be analyzed entirely in chapter 6 followed by the discussion in chapter 7. Lastly, a conclusion of the thesis is drawn to see if the thesis has brought a benefit to the industry or if the results could be improved with the knowledge gained during the process.
2 Method

In this section, information about the scientific methodology is given. It will give the reader more detailed information about what to expect from this work.

2.1 Scientific Approach

To answer the research questions mentioned above, an evaluation of different methods is going to be conducted. To evaluate the approach of using LD_PRELOAD for dynamic analysis, a proof of concept is going to be implemented.

2.2 Method Description

To be able to answer research question one and two, various sources will be consulted. These sources are scientific articles, documentations, as well as experiment results from researchers and the work of this thesis. The self-conducted experiments must be unified and comparable. Therefore, a fresh emulator with the same specifications is going to be created for each dynamic analysis method. Furthermore, the same applications will be set up and evaluated for every method. Together with the supervisors of this thesis, it was agreed that the applications to be tested should be of the banking sector. The reason being that banking applications are intricately linked to the usability, comfort, and privacy of the user’s bank account. The fact that financial applications usually have a lot of security mechanisms implemented that thwart debugging an reverse engineering will be considered in the further chapters. The selection was further narrowed by focusing on three Swedish banking applications, which have been anonymized. The different dynamic analysis methods that are evaluated will have a short conclusion in their respective chapter, while the entire analysis and discussion of the result will be done afterwards. Furthermore, one insecure application, namely “InsecureBankingv2” is included which is likely to be less protected and yield more interesting results [6]. The following applications will be tested with dynamic analysis methods to be introduced later:

Banking Application One - BA1
Banking application Two - BA2
Banking application Three - BA3

It could be argued that the number of applications to be tested could be in higher number, but it needs to be emphasized that the purpose of this thesis is not to test multiple banking applications to find flaws. The focus is to demonstrate how different tools can be evaluated on other criteria, as can be seen in section 4.2. A problem to be solved is that different approaches have different requirements to the environment in order to function, such as API Level or root-access. This theoretical data from articles and documentations, as well as the empirical data, is qualitative and needs to be assessed in order to enable evaluation of dynamic analysis approaches (RQ1) and to keep track of related API calls (RQ2). For research question one, key criteria for the comparison of dynamic analysis must be defined to enable a valid comparison. Research question two will rely mainly on various documentations as well as program analysis. The results from research question one and two and additional exploration will be essential to solve research question three and four. Additionally, the previously mentioned type of sources is needed to retrieve useful results. To solve research question four, a proof of concept is going to be implemented.
2.3 Reliability and Validity

The findings of this thesis are based on two main resources. First, the results of the evaluation of dynamic analysis methods are split up into systematic literature review, as well as practical self-implementation. In order to receive objective results, the same applications will be tested based on the same metrics for every approach. Second, the research on literature will be reliable as long as there are no controversial publications of research that will collide with the current state of dynamic analysis methods. Furthermore, the factor of the quick aging of technologies is a constant factor in computer science and could affect the results, since Android’s architecture and structure could change in the future. The results of the self-implemented, more practical approach is that everything in computer systems is highly dependent on platform, software versions as well as the realization of the researchers themselves. Therefore, one can say that repeating this thesis could yield different results in the future and is not an all-time valid work. The grading of the metrics in the evaluation is solely based on observations of experiments as well as the mentioned consulted sources. It can be argued that these values are partially subjective or might change in time. Nevertheless, the validity of this project is justifiable, due to the structured procedure in which the thesis work has been executed and documented in this report. Moreover, the evaluation is dependent on literature review and the implementation, which will correlate with the results, thereby increasing the validity.

2.4 Ethical Considerations

This work has been done without the participation of research participants and therefore cannot harm anyone’s privacy or dignity. The persons mentioned gave me their consent in being mentioned or are public personalities. Furthermore, a realistic and transparent application of this research project is given and can be reviewed at any time. All information and results stated in this work is either cited or worked out by the author of this thesis. The applications, that have been used for research purposes, are publicly available and have not been reviewed for vulnerabilities; no sensitive information was leaked. The banking companies have been approached and had the chance to raise an objection but did not do so. Furthermore, the applications have been anonymized to prevent any form of misuse of this thesis.
3 Theoretical Context

To be able to understand and answer the research questions previously shown, it is necessary to be aware of basic Android related technologies and the Android architecture as well. Moreover, it will act as a knowledge base to make later chapters better understandable.

3.1 General Android Architecture

Figure 3.1 visualizes the architecture of the Android platform. The purple-marked fields contain Native C/C++ Libraries. Components and services can be written in native code for various reasons, such as graphical library support or performance. Therefore, these components require C or C++ libraries to function. Some applications require the use of C or C++ libraries, therefore, Android provides the Native Development Kit (NDK),
which makes it possible to implement parts of an application as native code. The green-marked area represents the Java Application Programming Interface (API) framework. The majority of Android applications are based on a Java-based implementation. That is because Android enables all features of the OS to be accessed via Java APIs. In the area highlighted with yellow, the Android Runtime, (ART) is shown. Every application running on Android devices is running on its own Android Runtime. Prior to the Android version 5.0, the predecessor Dalvik was used. The main change from Dalvik to ART was that Android introduced the Ahead-Of-Time (AOT) compilation which means that the byte code is compiled to native code when the application is installed. In comparison, Dalvik used the just-in-time (JIT) compilation where the bytecode is compiled to native code when it is needed. Since Android 7.0 ART supports both AOT and JIT and combines both advantages. The major changes from Dalvik to ART can be summarized in three points. First, AOT enables multiple virtual machines to be run on memories that have low memory capabilities since applications do not have to be compiled during runtime. Second, no initial compilation is needed when installing an application which increases performance. The introduction of hybrid AOT and JIT usage increases performance even more. Lastly, the memory management has been improved by creating a more intelligent memory allocation and garbage collection.

The Hardware Abstraction Layer (HAL) in turquoise consists of modules for providing high-level interfaces for different types of hardware such as cameras, speakers, or microphones to be used by applications. The device driver itself is located in the kernel. The Linux Kernel is the foundation for the Android Framework. The Android Runtime relies on different functionalities of the Kernel, mainly in the terms of memory management, device drivers, the networking stack, file system I/O et cetera [7]. Furthermore, Linux offers a well-tested security concept that will be described in the following sections.

3.2 Android Security Architecture

Android devices can be used for many different purposes, from streaming videos, browsing social networks, online banking, and much more. Therefore, security mechanisms must be in place to protect user devices, data, and identities. On the official website where Google provides technical information about Android, five key security features of the platform are listed [8]. These features are:

1. Robust security at the OS level through the Linux kernel.
2. Mandatory application sandbox for all applications.
3. Secure inter-process communication.
5. Application-defined and user-granted permissions.

3.2.1 Kernel Security and Application Sandbox

The foundation of the operating system is a Linux Kernel. It has been used for many environments and is therefore well researched and accepted by the security community. It comes with key features such as process isolation, user-based permissions model, and advanced Interprocess communication (IPC) model as well as an application sandbox. The application sandbox isolates every application from others via user ID’s and runs them
in separate processes, which makes memory corruption attacks less dangerous, since the scope is only application based. This does not erase the possibility of attacks that can lead to the breakout of the sandbox, as Google states themselves in their documentation: "Like all security features, the Application Sandbox is not unbreakable. However, to break out of the Application Sandbox in a properly configured device, one must compromise the security of the Linux kernel." [9]. Note that Java is not affected by the possibility of memory corruption since it is considered a type-safe language. By default, applications cannot interact with other applications and have very limited access to the operating system. The application sandbox is located in the kernel and is therefore based on native code.

3.2.2 Interprocess Communication (IPC)

IPC provides developers with system functionalities for a more secure interprocess communication between the separate sandboxes.[10]. It enables information sharing, privilege separation, and data isolation. IPC in Android can be implemented using programming mechanisms, such as Intents, Binders, Messengers, Services and Broadcast Receivers. Android IPC mechanisms enables the verification of an application that wants to perform an interprocess action which increases the degree of integrity [11].

3.2.3 Application Signing and Permission Model

Every application that runs on the Android platform must include a signature from the author. It enables users and developers to verify any application. Unsigned applications are being denied access to installing on a device, either via Google Play or the Android package installer. According to Google, the signing process increases the trust between developers and Google. The developers can be sure that their application is distributed exactly like they intended, which also makes them accountable for the behavior of the application. As described earlier, applications in the Android Sandbox have restricted access to system resources. Still, applications need to use different system resources, such as camera functionalities, Global Positioning System (GPS), telecommunication, and more. To ensure safe access to these functionalities, Google introduced protected APIs. Users that want to install an application get a prompt listing the functions that the application will use and can decide if they want to continue the installation or not. The required functions of the applications are set in the application manifest file by the developer [12].

3.3 Development of Android Applications

Commonly, applications are developed in Java, since it is the best supported language from Google. The development environment Android Studio marketed by Google provides the developer with many useful features in order to develop Android applications. Besides writing applications in Java, there are possibilities developing hybrid applications. Hybrid applications are written like regular web applications as a combination of JavaScript, HTML, and CSS, which are placed inside a native application that provides a WebView. The WebView can be imagined as a borderless browser window. Having the permissions of an application, hybrid applications can access device functions such as the camera, GPS location, contacts, and more. Applications that rely on graphical resources are often developed in native code, since they can utilize the Open Graphics Library (OpenGL) and do not rely on graphical elements, such as buttons, provided by Java. It happens that developers decide to deploy specific functionalities in native libraries, even
though the core application is written in Java. There are solutions like Xamarin[13], that allow the developer to write an application in a different programming language such as C#.Net and to deploy them independent of any platform, for example Android or iOS. No matter how an application is developed, it will be bundled in the APK format, which will be explained in the following chapter [14].

3.4 Structure of an APK

Android application Packages (APK) are package file formats used on the Android platform to enable a smooth distribution and installation of Android applications over the internet. An APK bundles several important files that are needed to install and run the application. As previously mentioned, different approaches exist to develop mobile applications that result in a different content of different APKs. Since the applications in this thesis are Java based, the main components of this kind of applications are presented. Important for this thesis is that it contains class files with the .dex file extension. That means that the class files have been compiled into a Dalvik Executable (DEX) which is an understandable format for the Dalvik or ART virtual machine. Android applications are mainly based on Java and they are compiled to byte-code in order to make it understandable by the Java virtual machine. To be able to run the application on the Android runtime, this byte-code gets translated in the previously described DEX files.

3.5 Definition Dynamic Analysis

To evaluate dynamic analysis methods, it is first imperative to define the term “dynamic analysis”. Besides static analysis, dynamic analysis is a core concept on how software can be investigated. Ankita Kapratwar, Master graduate at San Jose State University, described the term dynamic analysis in his master’s project “Static and Dynamic Analysis for Android Malware”[15] detection the following: “Dynamic analysis is a detection technique aimed at evaluating malware by executing the application in a real environment.”. The general difference is that in static analysis, the code to be inspected is not executed, while dynamic analysis is doing exactly this, either by executing instructions in real or emulated environments. Sometimes the border between static and dynamic analysis becomes blurry, as described later in the report. The focus of this project is in the evaluation of dynamic analysis rudiments and will not provide deeper information or investigation of static analysis.

3.6 Type-Safety

Further, in the report, the term type-safety is going to be used. To understand what is meant by saying that a programming language is type-safe, C and Java are being used to display the difference; Java is said to be a type-safe programming language while C is not. A suitable definition of type-safety is the following: "In a type-safe programming language [sic] any variable with a declared type will always reference an object of either that type or a subtype of that type.". An example is if a method expects a variable of the type double and an integer is passed. In Java, this would result in a compiler error, which means that the program will not be able to run. In C, it will lead to a type error and undesired program behavior.[16].
4 Evaluation Of Dynamic Analysis Methods

In the following section, different dynamic analysis methods will be evaluated along with scopes and metrics, which are presented in the next section. In the end, a conclusion will be reached.

4.1 Scope

In the wilds of the internet, there are hundreds of possible dynamic analysis methods, such as tools, frameworks, virtual machine images, and more. Ranging from full source-code reviews to black-box testing, and from static analysis, to dynamic analysis approaches. Frida.re, one of the most developed tools, is a scripting framework that offers support for different platforms but requires self-implementation and can only be used with basic features immediately after installation. Another group of tools are automated web analyzers like Tracedroid. They perform automated analyses in an emulator after uploading an APK file but are limited to certain basic interactions. Another approach is based on LD_PRELOAD, which is an environment variable on the Android platform that specifies additional shared objects to be loaded before others by the dynamic linker. In section 4.6, a deeper introduction of LD_PRELOAD is given. In cooperation with TrueSec, the decision was made to include an implementation of LD_PRELOAD into the thesis for mainly two reasons. The first reason is that TrueSec saw the potential in such an application for themselves. The second reason is that LD_PRELOAD, unlike other approaches to be presented, is not a product, but a component of the Linux operating system and must be implemented. Therefore, the prototype application in section 5 was developed as a proof of concept in order to show the potential of this method. There are hundreds of other tools, but analyzing the majority of them would be out of scope, as these methods had to be narrowed down to the most relevant ones. This was a difficult task, since the landscape of tools is changing rapidly. Some tools that had support one or two years ago might not support all current technologies, which should be tested as new tools appear frequently. This became clearer reading an article “Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation” in which different tools were analyzed [5]. These tools were cross correlated to the "Collection of Android security related resources" found on two Github repositories ; "Android-Security-List" [17] and "Android-Security-Awesome" [18]. These Github repositories are updated frequently and were recommended by the supervisor of this thesis, Philip Åkesson. With these two sources, the most relevant tools and methods were chosen to be evaluated. Hence, it is possible to get an interesting result without including outdated or inconsistent approaches. The output of this process are the following 5 dynamic analysis categories:

1. Dalvik/ART Debugger (Java)
2. Native Code Debugger (C/C++)
3. Web Analyzers
4. LD_PRELOAD
5. Frida

In the following sections, the analysis of the mentioned approaches will be introduced and evaluated.
4.2 Metrics

The following metrics are defined to make an evaluation of different dynamic analysis approaches possible. Based on the score of these values, a foundation is given to compare them in an objective way. Each criterion will be given a score from one (not fulfilled the criteria) to five (completely fulfilled the criteria). These metrics were developed in cooperation with the software security testers of TrueSec located in Malmö.

**Support:** Some solutions found are made by private persons and the only possibility to get support is to open an issue on Github in the hope of an answer. It varies if and when people are replying. Other solutions are provided by companies who offer a professional service with hotline and support email addresses.

**Customization:** Different approaches in dynamic analysis vary heavily in how much the software tester can customize or affect the run and output of a test. This can affect the need and choice of a certain tool.

**Usability:** The installation and configuration of dynamic analysis solutions vary heavily. Some are up and running in a few minutes after starting to work on them, while others need special dependencies and configuration. Since software security testers are usually high paid experts the time to set up and use a testing environment is an important factor.

**Automation:** Automation of tests can save time, which has a direct impact on the cost of a test. If a dynamic analysis solution is or offers the frame to be automatable, it can increase the efficiency of any tester using it. A well automated solution also empowers the scaling of a test, which is interesting when it comes to testing more applications.

**Costs:** In this evaluation, cost is defined in the sense of time consumption for the security tester and therefore the amount he needs to be paid to perform the test. Furthermore, whether a solution is free or commercial will affect the cost score. The lower the cost, the higher the score.

4.3 Dalvik / ART Debugging

The previously mentioned Android Runtimes ART and Dalvik run DEX Bytecode on the Android Platform. Since Java is the most widely used programming language of Android applications, it is necessary to analyze DEX code in an exhaustive application security review. To improve this process debugging tools are used. It is important to take in consideration that the Android Debugging API is changing with the development of Android itself to add additional functionality. Debugging an application using older Android version will miss some of the newly added debug features. Debugging enables the tester to set breakpoints and step through the code. A breakpoint signals the debugger to stop at the chosen line of code. The tester can inspect values of variables or methods calls to find logic flaws or other sources of unexpected behavior. This method is not only interesting for software developers improving their code quality and correctness but also for code security analysts. Code security analysts can use debugging in order to find potentially dangerous sections in the source code and report or fix them. A problem with this approach is that the source code is not always available to a code security analyst. How to get the source code from an arbitrary APK file is described in the Experiment section. The following section will discuss and evaluate how effective Dalvik/ART debugging is
and how it can be used in order to find security violations. Some might argue that this method is not pure dynamic analysis, since extracting source code from an APK file can be seen as a starting point in static analysis. On the other hand running the modified application and debugging, it falls into dynamic analysis so the approach to be described in the following sections can be seen as a hybrid analysis and should not be missed in this report [19][20].

4.3.1 Setup

The setup for testing the five applications mentioned above was an Ubuntu 16.04 host machine. The toolset includes Apktool, dex2jar, JD-GUI and the AndroidStudio IDE. The emulator is a Nexus 5, which runs on the API 22 running on an x86 processor. For accessing the emulator the Android Debug Bridge (ADB), provided by Google, was utilized. The ADB is described the following way on the developers page of Android: The adb is a versatile command-line tool that lets you communicate with a device (an emulator or a connected Android device). The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device."[21]. To get started, the application had to be decompiled to ensure that the debuggable flag is set in the manifest file of the application. Besides various permissions and other settings, the debuggable flag can be found in the application tag of the AndroidManifest.xml as shown in the picture below, as long as the value is specified. To be able to read the manifest file from an application, it is not as easy as simply unzipping the APK, since the manifest file is binary encoded. Therefore, the program Apktool, or a graphical solution like JD-GUI, can be used. In this experiment, the command line application Apktool was used with the following command:

```
apktool d app_test.apk && nano app_test/AndroidManifest.xml
```

Apktool is ran with the option "d", which, besides disassembling the sources, also decodes the manifest file. With a text editor, it is possible to analyze the manifest file and if needed to set the application flag to debuggable = “true”. Further modification of files can be performed. To apply the changes, the sources have to be repacked. This can also be performed with the help of apktool issuing:

```
apktool b -f -d app_test
```

Now that the application is set to debuggable, the actual extraction of the sources files can be performed. To do so, the apk-file has to be transformed into a jar-file. This can be done with the tool dex2jar. By issuing the following command, a jar file called app_test-dex2jar.jar was created.

```
d2j-dex2jar.sh /paypal_test/dist/app_test.apk
```

Figure 4.2: Example of an Android manifest file snippet

![Android manifest file snippet](image)

12
To retrieve the actual source files this jar file, needs to be decompiled. Java decompilers like JD-GUI or others are freely available on the internet. In the graphical user interface of JD-GUI, the decompiled sources can already be seen and analyzed. Furthermore, the files can be saved in a zip file, which contains the project structure, including readable Java files. After changing the needed parts of the source code, the sources can be repackaged using Apktool. To be able to install the changed APK on an Android device or emulator, it must be signed with a public key. If the tester does not have a valid key yet, it can be created with the Keytool in the following way:

```
keytool -genkey -v -keystore myAPKkey.keystore -alias myAlias -keyalg RSA -keysize 2048 -validity 10000
```

Lastly, the application needs to be signed with the created key.

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore myAPKkey.keystore app_test.apk myAlias
```

To perform the debugging, Android Studio is used. It integrates the ADB functionality that does not require any command line interaction. The tester simply has to set breakpoints at the places in the code that he wants to investigate or, in case of a full code debugging session, at the program entry point.

### 4.3.2 Experiment

Before describing the experiments more in-depth, it should be mentioned that the decompilation only applies to Java code. If the application contains native code, the previously described approach is complicated as stated in the article "During our research, we encountered multiple applications calling methods from native libraries, causing the aforementioned problems" [5].

**BA1** The decompilation with Apkstudio proceeded without any problems, unlike the compilation. In the AndroidManifest.xml, it was possible to see that the application was created for Android version 4.4.2, with API level 19. Resource files where not defined as well as other errors, which made it pointless to pursue further attempts due to the restricted time-frame. The recompilation resulted in multiple errors with the following pattern:

```
bal/res/values/public.xml:531: error: Public symbol drawable/# here is not defined.
```

or

```
ERROR: 9-patch image bal/res/drawable-mdpi-v9/gray_tab_button.9.png malformed.
Must have one-pixel frame that is either transparent or white.
```

It seems that the mapping of graphical resources is a problem in the recompilation process.

**BA2** There were no issues with decompiling BA2. The API level was defined in the AndroidManifest.xml with 23, while the version number is 6 with the code name Marshmallow. The debuggable flag was not set in the AndroidManifest.xml so it needed to be added. Afterwards, the recompilation process worked out with no problems as well.
With the help of JD-GUI, it was already possible to glimpse into the file structure of the application.

It is possible to see that the classes, methods, and variables are not obfuscated, which simplifies a manual static analysis. Storing these decompiled sources revealed that it is a total of 3589 items. Importing the files into a new Android project was not a straightforward process, since missing dependencies and the file structure differs to a great extend. Moreover, the sheer number of files made it difficult to place every file in the right spot. The time that it would take to get the application to a working state would exceed the time frame of this thesis.

### BA3

The decompilation with apktool worked without any problems. Firstly, the API level, as well as the version number were retrieved by investigating the AndroidManifest.xml file. BA3 application uses the API level 23 with the Android version 6, codename Marshmallow. It was furthermore no problem to add the debugging flag and recompile the application. The recompiled application contained 4991 files and exceeded the number of the previously analyzed application. In contrast to BA2, the application was obfuscated. Class, method, and variable names were substituted with letters such as 'abc', which made it more problematic to understand the source code. An example class can be seen on figure 4.4. Similar to BA2, it was not possible to create an Android project without a considerable amount of compilation errors. To piece the information together in order to have a working project, it would take a high number of hours, resulting in an expensive operation. Therefore, it was not further pursued and the current state has to be considered as a result.

### InsecureBankv2

InsecureBankv2 is published by the Github user dineshsetty and is described by him as a “Vulnerable Android application for developers and security enthusiasts to learn about Android insecurities”. InsecureBankv2 was part of the presentation list of the 2015’s and 2016’s black-hat conference. The AndroidManifest.XML defines the applications API levels from 11-24, which belong to Android version seven, code name Nougat. It covers a big list of vulnerabilities, which in detail can be found on the application’s Github page. Since the application is published on Github, its sources are publicly available, which makes the preparatory process of getting the sources unnecessary. To perform a white-box test, the source-code is required [22]. The import of the project into Android Studio was yielding no errors. Now it was possible to set breakpoints in the spots that can be interesting to see the current values. Moreover, Android Studio
detected the usage of deprecated methods, which might be exploitable and therefore aid the developer preventing these vulnerabilities. Screenshot 4.5 shows the login class with the method `postData`. In the bottom of the window, it is possible to see the current state of variables, which enables the tester to investigate the program flow. The tester is enabled to step through the program execution, watching how values change, what methods are called, and what instances of variables are being invoked. The first thing to notice is that a test account is present in the comments, which could be used to log in. Furthermore, there are several deprecated methods present in the `postData`. It is necessary to note that this work is not a code review and will not debug the whole application.

4.3.3 Application for Dalvik/ART debugging

Java plays a major role in the Android application environment. Since mobile applications tend to become more complex, it may be troublesome for developers to keep their systems completely free from bugs and errors. While static analysis is a common practice, it has some disadvantages compared to dynamic analysis. A big factor is time. Going through hundreds or thousands of lines of code can be avoided by **hooking** into the application and investigate the behavior during runtime. Another disadvantage static analysis has is that vulnerabilities that occur during runtime can not be detected, since the interaction with the machine is not included. Dynamic analysis offers this opportunity, since it includes every factor that occurs during a normal execution e.g. user interaction and network connections. A debugger utilizes the advantages of the dynamic execution by setting breakpoints at interesting lines of code, saving time reading irrelevant source-
code. Thus, a developer can reveal logical errors, inspect program states, or change the behavior of the program flow at runtime. The data flow becomes more comprehensible, hence debuggers are an essential building block in the security industry. Although, having the source-code present when testing seems to be significantly efficient, as seen in the previous experiment. Reverse engineering an application is a tedious and sometimes impossible challenge depending on the given time frame and the expertise of a software security tester [23]p.205-212.

4.3.4 Results

Unfortunately, the de- and recompiling attempts of the three Swedish mobile banking applications were not available for dynamic testing. Notable is that the decompilation does not apply to any native code in the application. Even if all Java code is recreated in a readable format, it may not be enough to figure out all functionality of the application depending on the relation of native and Java code. Nevertheless, it was possible to retrieve the source code from the APK file, which enables a software security tester to perform a static analysis. The evaluation will be based on the conclusions and observations on the vulnerable banking application.

Support: The documentation of the ADB, as well as the quality in which the debugging functionality is built into Android Studio, holds the standards users expect from a company like Google. Furthermore, there are multiple support forums where active discussions are held. The score gets reduced by one point since a developer has no direct contact opportunities to get support e.g. hotline.

Customization / Extensibility: The debugging process is not customizable insofar as the terms being able to extend the debugger to perform different operations. Still, the tester can perform his tests differently, e.g. placing of breakpoints or investigating nested
objects. Moreover, Google ships new debugging functionalities when updating the API that extends the features of the debugger.

**Usability:** The integration of the debugger in Android studio is exceptional. The tester simply has to set the debug flag to true and place the breakpoints where they should be. By pressing the debug icon in the Intelligent Development Environment (IDE), Android Studio automatically opens a window, where it is possible to select the emulator, and starts the application in there. The debug view enables the tester to investigate variables and step through the program.

**Automation:** In general, debugging Java code is a highly manual task that involves the experience and knowledge of a software developer. Nevertheless, the authors of the article "Using SPIN for automated debugging of infinite executions of Java programs"[24] found a way to automate parts of the debugging process. However, debugging is a highly reactive task that involves machine learning. In the paper “Are Automated Debugging Techniques Actually Helping Programmers?” Chris Parnin and Alessandro Orso discuss the effectiveness of automated approaches, but argue that debugging is a highly reactive task which requires machine learning to train a machine mimicking realistic human input.[25].

**Costs:** The experiment revealed that depending on the project size and the grade of obfuscation, debugging can be very time-consuming. It needs to be evaluated by the tester and the contractee if an application is testable in a reasonable amount of time. A factor can be if the test is executed as a white box or black box test.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>4</td>
</tr>
<tr>
<td>Customization</td>
<td>2</td>
</tr>
<tr>
<td>Usability</td>
<td>4</td>
</tr>
<tr>
<td>Automation</td>
<td>2</td>
</tr>
<tr>
<td>Costs</td>
<td>1</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>13</strong></td>
</tr>
</tbody>
</table>

Table 4.2: Evaluation Results of Dalvik/ART debugging

### 4.4 Native Code Debugging

Besides Java, Google offers support for implementing parts of native code in an application. Often, native code appears in the form of shared objects. As previously described, the Native Developer Kit (NDK) was developed to simplify the needs of developers. Therefore, the probability of finding classes written in native code is higher, and the security review needs to be expanded to not only Java, but also these native classes. Note that not every application contains native code. That makes security reviews more complex, since either a new tester with knowledge in C or C++ needs to do that part of the review has to be consulted, or the usual software security tester needs to be educated in order to perform sufficient tests for the native code base. The methodology of debugging native code is used to test an application. This section is dedicated to show how native code can be debugged on an Android emulator without having the actual code base.
4.4.1 Experiment

For creating a comparable result to the Dalvik/ART debugging test, the same applications will be used for testing purposes. On the developer page provided by Google, the flow of incorporating native code is described[26]. This flow can be found on the developers-page. One step states that NDK can be used to compile the native (.so, .a) libraries. Therefore, it is possible to see that native code exists in an application by unpacking the APK file and look if .so or .a libraries exists. Another way is to examine the DEX code and look if there are any Java Native Interface (JNI) calls. The JNI is the interface used for the communication between Java and C/C++ components. By knowing that the first step was to examine if the decompressed APK files contained any .so files. The results can be seen below:

4.4.2 GNU Debugger (GDB)

For debugging native code on Android, GDB is used. GDB is a debugger published by the GNU Project. It has a command line interface and its four main functionalities are described on their website [27]. They are:

- Start your program, specifying anything that might affect its behavior
- Make your program stop on specified conditions
- Examine what has happened, when your program has stopped
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another

To use GDB on a mobile Android device a server has to be started on the mobile device. The tester then is able to connect to. The setup of this experiment consists of 3 major steps. Firstly, the GDB server has to be set up on the emulator, secondly, he connection from the client to the GDB server and lastly, the actual debugging. To set up the GDB server on the emulator, the file system has to be mounted with read and write permissions to make it possible pushing the server on the device. This can be done by issuing:

```
adb shell mount -o rw,remount /system
```

Afterwards it was possible to push the server via:

```
adb push /Android/Sdk/ndk-bundle/prebuilt/linux-x86_64/bin$ ./gdb
```

Attaching the debugger is done with the command where #pid represents the process to intercept. And the port number has to be the synchronized with the receiving client on the host system. In this case the port number is represented with #portNumber.

```
gdbserver :#portNumber --attach #pid
```

To guarantee that the packets are send from the emulator to the host machine the TCP traffic needs to be forwarded. To do so “adb forward tcp:#pid tcp:#pid” can be executed on the host machine. Performing these steps leaves the emulator in a read to test state. The Android SDK includes a precompiled GDB client depending on the architecture. To access the GDB console it is necessary to run it. The client can be found, depending on the architecture at /Android/Sdk/ndk-bundle/prebuilt/linux-x86_64/bin/gdb. In the GDB
console the tester now can connect to the GDB server which has been set up before via target remote :#portNumber. This leaves the tester in an environment where he can debug the programs natively [28].

4.4.3 Application for native code debugging

Developing native code for Android applications involves insecure type and memory handling, while Dalvik/ART offers memory security, and developers do not have to be worried. This has to do with the previously explained type-safety. This leaves native code open for more serious vulnerabilities, since a potential attack could have the memory as an attack vector. To minimize the surface, developers can use a debugger to find code flaws, while attackers and security testers can use debuggers to gain information from the instructions in the application. Native debugging should also be part of a security testers skill-set for specific cases where debugging Java code fails. An example could be that the Java code is obfuscated and/or encrypted. While debugging the Java code would yield no result, the decrypted code could be analyzed in the memory using a native debugger. There are several tools that extend functionalities of general debuggers to aid finding specific flaws. An example for that is the GDB exploitable plug-in from Jonathan Foote [29]. These tools help the user to determine if the crash of an application could be a security issue. Besides the mentioned memory safety issues, the previously mentioned vulnerabilities in Dalvik/ART debugging can be found using a debugger [23] p.205&p.221.

4.4.4 Results

After setting up the test environment the decision was made to not look deeper into the actual debugging process from a practical perspective. This had the reason, that in order to perform a sufficient native debugging session more knowledge about the usage of the GDB, as well as the assembly language was needed. After consulting the official GDB manual the scope of performing a sufficient test with GDB would exceed the scope of this thesis and the capabilities of the author is not conducted [27]. This does not indicate that setting up, and researching about native code debugging was not yielding to any results. The main advantage of debugging native code is, that the software tester is able to explore system services, Random Access Memory (RAM) and network usage on a low level [30]. It is furthermore possible, to step through critical instructions, such as logins and connection creations.

Support: Google offers the ndk-gdb, which enables the tester to launch a native debugging session. Still, the official support does not offer how to setup the debugging environment with the GDB server on the Android device. The documentation on the GDB page does not mention crucial information needed to get a remote debug session running on an Android device. To set up the GDB in this project, third party websites were consulted.

Customization: In terms of customization, debugging native code with GDB is comparable to the previously evaluated and described Dalvik/ART debugging process.

Usability: Usability-wise the debugging of native code was considered very advanced. Performing a consistent debug process requires deep knowledge of C and Assembly code with respect to the architecture below, which in the case of Android is either ARM or
x86. The setup was mainly possible due to the blog post described previously. Navigating through the GDB shell and debugging native code is something a security tester is not able to learn in a short amount of time. This conclusion can be made by reading the documentation of GDB.

**Automation:** The native debugger is mainly used for specific problems and not for e.g. testing hundreds of applications for security flaws. If the tester knows what he wants to know, there are approaches to script certain tasks of a debugger. Nevertheless, during the research, several tools/plug-ins were found that offered automation for specific task, such as a memory leak detector [31].

**Costs:** The experiment revealed that in order to be able to perform a valid test in GDB, the testers must be educated or an expert must be consulted. Since automation is not a reliable option for every requirement, the test has to be taken out manually, which is time consuming and yields high costs for experts. Nevertheless, no commercial tools are required to perform manual debugging tests.

<table>
<thead>
<tr>
<th>Support</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customization</td>
<td>2</td>
</tr>
<tr>
<td>Usability</td>
<td>1</td>
</tr>
<tr>
<td>Automation</td>
<td>3</td>
</tr>
<tr>
<td>Costs</td>
<td>2</td>
</tr>
<tr>
<td>Sum</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4.3: Evaluation results of native code debugging

### 4.5 Automated Analyzers

The automated analyzers inspected in the research phase are web applications, where an APK file can be uploaded and tested in a virtual environment. Similar tools exists that can be installed on their own systems exist, but were not included in this chapter. Often, automated web analyzers consist of tools described previously and automated the process. There are tools that perform static and dynamic analysis; Regarding the scope of this project, static web analyzers have been disregarded. Automated analyzers have the main purpose of performing automated test out of the box, without further source code analysis of the tester itself. Collections of online analyzers were found in the previously mentioned Github repositories containing various android security relevant resources [17][18]. In this thesis, the focus is on free solutions. Moreover, the automated analyzers had to conduct not only static analysis, but dynamic analysis, which excluded several tools of the initial list. Including dynamic analysis is extending the scope of the test by simulating common behavior of a phone, such as phone calls, and message transaction at the cost of a more complex and case driven need for implementation.

The candidates for testing are:

1. AndroTotal
2. Dexter
3. Tracedroid
4. Visual Threat
It was necessary to filter the list of fifteen products for the following reason; Solutions that offer solely static analysis are commercial without trial or are outdated. The result of this process is the following, heavily reduced list of items. The candidates for testing are:

1. Tracedroid
2. Visual Threat
3. NVISO ApkScan
4. Frauenhofer App-ray

After sorting out unrelated candidates, it can be seen that the majority of tools do not have dynamic analysis implemented. The remaining tools are investigated in the following section.

### 4.5.1 Tracedroid

Tracedroid is a dynamic analysis tool developed by VU Amsterdam and maintained by Victor van der Veen and Christian Rossow. “Tracedroid records the behavior of the executed application, such as its network communication, the user interface (UI), but also its internal function calls and Java code that is executed. To trigger the [sic] application real behavior, Tracedroid emulates a few actions, such as user interaction, incoming calls and SMS messages, etc.’ Like the other automated analysis tools, the user only uploads the APK file to be tested. The output consists of different log files including logcat log files, network-dumps, call-graph visualization and more. Logcat is a command-line based tool that dumps system messages and stacks traces of errors that have occurred. The analysis log file contains the coarse-grained steps of analysis. In that log file is described what interactions with the application are simulated, e.g. phone calls, incoming messages, outgoing messages, and more. Detailed information can be seen in the other log files. Another feature is the creation of a call graph which has limited use, since it is likely to become unclear if the application is growing [32].
4.5.2 NVISO ApkScan

NVISO is a security consultancy located in Brussels, Belgium. NVISO is working on ApkScan, which is mainly focusing on malware detection. This application is currently in beta state and mixes static and dynamic analysis. Unfortunately, not too many technical details are publicly available about the testing procedure. The test devices are running Android 4.1 Jelly Bean (API level 16). The output consists of a report covering multiple static analysis results besides the dynamic analysis output. The dynamic analysis results contain file interaction of the application, Network activity, incoming and outgoing phone calls, information leakage, and cryptographic activity. Another interesting feature is the creation of a Graphic Interchange Format (GIF) that shows random artificial input that is supposed to mimic human interaction [33][34].

4.5.3 Visual Threat

VisualThreat is a connected-car security vendor based in Silicon Valley. They offer the service of uploading an APK, which is limited to fifty Megabytes, that is going to be analyzed and reported. The only information about the tool was a manual, which unfortunately was only available in Chinese. The report itself shows that a dynamic analysis is performed.

4.5.4 App-Ray

App-Ray is a commercial product by the APP RAY GmbH located in Vienna, Austria. According to their website, it is capable of dynamic and static analysis. In a report of the Fraunhofer Institute for applied and integrated security the tool is described in great detail by the core developers. The application is executed in an emulated environment and filters potential privacy breaches, user tracking, monitoring file access, as well as tracking information flows. Furthermore, it combines static and dynamic testing in order to extend the test coverage. Unfortunately, getting a trial version of App-Ray was not possible, and therefore it could not be tested [35].

4.5.5 Experiment

As done with previous dynamic analysis method, the four banking applications were briefly tested with all automated analyzers to see which results each of them yields. The combined results will be the foundation of the evaluation.

BA1 Submitting the APK of BA1 at tracedroid.few.vu.nl resulted in a failure. Trace-droid did not provide information as to why the application analysis failed. The only feedback that the user gets is that the test failed, and timestamps. The hash values can be ignored, since they can only be used to query for the report of choice. The output can be seen on the following screenshot. The result of Apkscan was also rather inconclusive. Besides a small amount general and static analysis results the dynamic analysis failed. Unlike Tracedroid the report of Apkscan offers a possible reason for the failure. The message can be seen on the screenshot below. The report created by visualthreat.com is web-based and includes general information about the APK, as well as information about the certificate with which the APK is signed. Sadly, the report is limited to static analysis. In order to get a more in-depth report including dynamic analysis, it is necessary to contact VisualThreat.
The analysis with Tracedroid succeeded and the report could be downloaded as a compressed archive. The content were 32 files that were mainly adb logcat dumps, but also different logs, a network capture file, and a call-graph of the application in a PDF. In the analysis.log file, the whole testing process is shown, which reveals the structure and methodology of Tracedroid. It is possible to see that Tracedroid creates an emulator, installs the APK, and starts logging the whole system. Then it simulates various interactions with the device as well as events, such as turning the network connection off or a low battery. BA1 gets also more specifically tested by simulating activities and services. The tester has the opportunity to investigate the various dump and log files to find potential flaws or malware. In comparison to BA1, the dynamic analysis of NVISO ApkScan did not fail. The user gets a quick overview of which files and network connections the BA1 uses and gives the opportunity to detect potential unwanted actions. The test could not place automatic phone calls or send SMS messages. Furthermore, no information leakage was detected. For more details, the tester could download the ADB logcat file to dig deeper.

Likewise, the BA2 APK Tracedroid was not able to process the BA3 APK. Also, NVISO ApkScan was not able to perform a dynamic analysis.

The insecure banking application was also not able to be tested by Tracedroid, and unfortunately, it was not obvious why. NVISO ApkScan was able to track file and network activity. The amount of information is still very limited.

4.5.6 Application for automated web analyzers

The scope on what automated analyzers can be found depends on the product itself. The main focus of automated analyzers is performing static analysis. Dynamic analysis is a niche functionality. Tracedroid and ApkScan performed usual interactions with the
device to show if the application would react in a certain way and logged it. Studying the report/log can show if the tested application was performing unwanted activities. This is especially interesting to find out if the application can be classified as malware. To show an example of what vulnerabilities automated analyzers can find, the report of the Frauenhofer Institute for applied and integrated security will act as an example. It states that the application is tested for four issues. The first one is if the application uses Transport Layer Security (TLS) and communicates solely over HyperText Transport Protocol Secure (HTTPS). Second, if TLS is properly implemented and e.g. does not accept certificates without validation. The third point is profiling. Profiling can be an issue if third-party libraries are in use that send out information, such as crash reports. If an application has the permission to read sensitive information, this could yield a privacy violation. Lastly, App-Ray checks the permissions and if the application uses advertisement libraries, which also could be a problem regarding privacy [35].

4.5.7 Results

After testing different automated products, the results will be presented with respect to the metrics defined. There are different factors within the pool of automated web analyzing tools that determine the usefulness for different application purposes.

Support: Support varies for different products. Considering that Tracedroid is a university project, it offers solely email support, while NVISO, VisualThreat and AppRay are products developed by the respective companies. The companies mostly did not provide any information about the methodology of the automated tools. Many tools that can be found are not maintained properly or are heavily limited in their supported file size.

Customization: The tester has no chance of in affecting or expanding the test, since the only point of contact is the upload of the APK.

Usability: In terms of usability, the web analyzers are easy to use. No expertise is needed to start the test. The simplicity of analyzing the results is different for the tools tested. All tested analyzers provided the tester with a report in the form of a PDF or collections of log files.

Automation: There are two forms of automation that have to be considered. One is the automation of submitting APK files to the web application, while the other is the automation of testing which occurs behind the scenes. Regarding the first form Tracedroid and NVISO ApkScan only enable the user to submit one APK files at the time while Visual Threat offers a solution to submit APK’s in batches. This functionality is not available in the free solution and could therefore not be tested, which highlights that at this point, automation could be improved. The internal automation details on how the different products operate where extracted from the reports itself. The complexity varies in between products the products tested.

Costs: The web analyzers are very time efficient considering the fact that the tester does not have to do much manually. The results are available withing ten minutes, depending on the state of the server provider. Since the tester is not actively required during that time, he or she is able to spend his or hers time more efficiently. As can be seen, some of the tools are commercial and offer trials or reduced features, while others are completely
free.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>2</td>
</tr>
<tr>
<td>Customization</td>
<td>0</td>
</tr>
<tr>
<td>Usability</td>
<td>5</td>
</tr>
<tr>
<td>Automation</td>
<td>3</td>
</tr>
<tr>
<td>Costs</td>
<td>3</td>
</tr>
<tr>
<td>Sum</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4.4: Evaluation results of automated web analyzers

4.6 LD_PRELOAD

The following chapter evaluates the usage of LD_PRELOAD based on the current state of implementation, but also realistic features that will be applied after this thesis project.

4.6.1 Introduction

To understand what LD_PRELOAD is, it is necessary to explain the dynamic linker in the Linux operating system. The dynamic linker is responsible for finding and loading shared libraries which are required by an arbitrary program; it prepares the program and executes it. A shared library is a file that contains source code that is used in multiple other projects and that can be loaded in order to avoid redundancy. Shared libraries are written in C and have to be compiled in a certain way to be executable on an Android device. This process will be described later. LD_PRELOAD is an environment variable in the Linux operating system that enables the issuer to load user specific shared libraries, that will be loaded before others. This can be used to override functions in other shared objects, and can be used to map frequently linked operations like read/write or to log encrypted communication sent through OpenSSL. Therefore, the tester is able to create their own shared libraries and hook onto hard coded paths, thanks to the load order. With this procedure, it is possible to override APK functions and manipulate calls without having access to the source code. Since LD_PRELOAD acts on the operating system level, it is not possible to miss out any interaction on the Linux file system. For an Android software security tester, this can be interesting, since it does not require the source code of the application to be tested [36][37][38].

4.6.2 Application for LD_PRELOAD

Since LD_PRELOAD is acting on the Linux operating system, and detects what the developer intends to detect, it is a very flexible approach, but it needs to be implemented from scratch. With the ability to hook into every low-level call of an application, the tested application is not able to interact with the operating system without being logged and analyzed by a tester. The scope on what to log is only limited by the system calls of the operating system and leaves much space for diverse implementations. This coverage is powerful, but generates a lot of data, as can be seen in the sections below. Therefore filtering and visualization is an important part for finding anomalies in the behavior of an application. A use case for LD_PRELOAD is to detect if a device is rooted. A rooted device can be problematic for application developers for different reasons. One approach for root detection is to investigate if certain superuser e.g. .su or .bin files are existent
on the file system. LD_PRELOAD can be used to detect if the application tested tries to interact with these files. Instead of reporting that the file exists, which could potentially cause the application to abort, LD_PRELOAD can be used to hide the existence of the super user file, the application will run, and can therefore be reverse engineered. Another possible use is to hook SSL functions so that SSL traffic from applications can be logged in clear-text. This would be more straightforward than messing around with Man-In-The-Middle proxy servers or adding new CA certificates to the OS trust store.

### 4.6.3 Compiling C/C++ code

For implementing native code parts into an Android application, Google offers the Native Developer Kit (NDK). Using native code is mainly used for performance improvement or for the sake of reusability of C/C++ libraries. The NDK offers a script named ndk-build, which has the purpose of invoking the right NDK build script, and creates a shared or static library or an executable to be used on the Android platform. The build script is called Android.mk. The Android.mk file is modular, where a module can be a static or shared library or a standalone executable [39].

### 4.6.4 Setup

The starting point was the prototype application presented in section 5, which currently has the purpose of logging file interactions. To find out if the application fulfills the basic requirement of logging file read, open and write operation, the application was compiled and pushed to the device. The potential of an LD_PRELOAD based implementation is based on the developer of the system. In the case of this project, the functionality is kept very basic. The following steps show the necessary steps to load the shared library on the device and performing preloading. First, it was necessary to remount the file system in order to make it read-and-writable to push the compiled file. To do so the following command was issued:

```
adb shell mount -o rw,remount /system
```

The shared object was pushed using

```
adb push sharedObject.so /system/lib/
```

To preload the shared library into a specific application, the command setprop followed by LD_PRELOAD and the own shared library:

```
setprop wrap.appname LD_PRELOAD=/system/lib/sharedObject.so
```

Issuing the command:

```
ps appname
```

enabled to find the process ID of the application of choice. With that process-ID querying for the running process and filtering by the pushed library name, it was clear that everything was in place. As described in section 5, the intercepted calls are being sent via UDP to be stored in a database.
4.6.5 Experiment

For performing tests on the applications, the current prototype described in section 5 is used due to the fact that LD_PRELOAD is not a product, but an environment variable. So far, the system has been implemented to intercept the following calls:

- **open()**: System level call from the UNIX operation system to open a file or creating a new file. `open()` returns a file descriptor which is used to interact with the file and finally close it [40] p.72.
- **fopen()**: Library function that builds on top of `open()`. `fopen()` is part of the standard C library (libc) which enables it to run on every device which contains libc.
- **__system_property_get**: A function that retrieves a system property.
- **__system_property_set**: A function that is used to set a specific system property.
- **fread()**: "The libc function fread() reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by pointer." [41].
- **fwrite()**: "The libc function fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by pointer."[42]
- **stat()**, **lstat()**, **fstat()**: These system calls retrieve information about a file. They differ in the following ways: `stat()` returns the information about a file with a name; `lstat()` does the same, but the named file is in this case a symbolic link and information about that link is returned; and `fstat()` also returns information, but based on an open file descriptor[40] p.279.
- **access**: The access function checks a user’s permissions of accessing a file.

**BA1** To gather data, BA1 was started, and some basic interactions performed. The results of the capture can be seen in the table below.

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>240</td>
</tr>
<tr>
<td>open</td>
<td>803</td>
</tr>
<tr>
<td>fwrite</td>
<td>0</td>
</tr>
<tr>
<td>fread</td>
<td>38</td>
</tr>
<tr>
<td>sys_prop_get</td>
<td>564</td>
</tr>
<tr>
<td>sys_prop_set</td>
<td>0</td>
</tr>
<tr>
<td>fstat</td>
<td>547</td>
</tr>
<tr>
<td>lstat</td>
<td>112</td>
</tr>
<tr>
<td>stat</td>
<td>248</td>
</tr>
<tr>
<td>access</td>
<td>134</td>
</tr>
</tbody>
</table>

Table 4.5: Function calls of BA1

It is clear that the application neither writes nor sets any system properties. The amount of calls can be analyzed in the basic web application to see if some suspicious or potential dangerous behavior can be seen.

**BA2** BA2 showed the boot screen when started. Afterwards, the application closed and kept a background process running. It was not possible to open the application in any way, which resulted in the application not being able to be tested with the LD_PRELOAD approach.
BA3 was started, and some user input was generated. No Login was performed due to a missing bank account. Still, a lot of data was intercepted. In the table below, the different Linux function calls can be seen with their respective number of calls.

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>113</td>
</tr>
<tr>
<td>open</td>
<td>467</td>
</tr>
<tr>
<td>fwrite</td>
<td>0</td>
</tr>
<tr>
<td>fread</td>
<td>37</td>
</tr>
<tr>
<td>sys_prop_get</td>
<td>383</td>
</tr>
<tr>
<td>sys_prop_set</td>
<td>0</td>
</tr>
<tr>
<td>fstat</td>
<td>391</td>
</tr>
<tr>
<td>lstat</td>
<td>61</td>
</tr>
<tr>
<td>stat</td>
<td>195</td>
</tr>
<tr>
<td>access</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4.6: Function calls of BA3

It is possible to see that with just a few user interactions, a lot of system calls are executed. In the current state of implementation, the tester can analyze what system and library calls have been intercepted. Figure 4.8 shows a small part of a list of JSON-objects that are simply displayed in the web application. The respective calls are logged with timestamps, a function name, and system call specific parameters, such as the file path of the application or a stream to a memory address.

Likewise, the other applications, also the insecure banking applications, were instructed with basic user input. The application provided example users, which were used to log in the user area. In the user area, it was possible to perform a money transfer and to change the root password. The amount of the respective function calls can be seen below.
### Results

The major difference by the other dynamic analysis methods and LD_PRELOAD is that LD_PRELOAD is not a ready-to-use program, and therefore needs a high degree of self implementation and deployment. On the other hand, the freedom and the fact that LD_PRELOAD acts on low level Linux functions leaves space for a variety of approaches testing mobile applications. The restriction to the Linux platform, the applicability to iOS, and Windows applications is not given.

**Support:** Since the implementation around LD_PRELOAD is not provided by any organization or company there is no official support or helpdesk. There is no support for the usage of LD_PRELOAD, besides the Linux documentation [43] and blog posts. Other solutions offered comprehensive tutorials on their web sites.

**Customization:** This approach is highly customizable and lets the tester intercept what he is interested in. The tester implements the server side part on the host as well as the client running on the phone themselves. The only boundaries present are boundaries of the operating system. The opportunities for further data processing and visualization are boundless.

**Usability:** LD_PRELOAD requires experience in the C programming language as well as the compilation process in order to write shared libraries. Furthermore, the shared library has to be deployed in order to run on an Android device. On the server side, any language that can receive and parse JSON objects via UDP is sufficient. As shown in the Implementation section of this report, several steps of self implementation are needed. With the current product, shown in section 5, the general usability is experimental but can be extended to a full scale web application.

**Automation:** By default, LD_PRELOAD does not offer any degree of automation. At the current state, the project has plenty of steps that have to be performed manually, such as compilation, deployment and execution. With the freedom to write scripts in every part of the system, nearly all steps can be automated, which leads to higher time efficiency for the tester.

**Costs:** It is hard to determine the cost factor, since the application around LD_PRELOAD was implemented in the scope of this thesis project. Since the starting point was literally nothing, the implementation is necessary to retrieve any results. Ignoring the implemen-

---

**Table 4.7: Function calls of the InsecureBankv2 application**

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>368</td>
</tr>
<tr>
<td>open</td>
<td>1443</td>
</tr>
<tr>
<td>fwrite</td>
<td>0</td>
</tr>
<tr>
<td>fread</td>
<td>50</td>
</tr>
<tr>
<td>sys_prop_get</td>
<td>1076</td>
</tr>
<tr>
<td>sys_prop_set</td>
<td>0</td>
</tr>
<tr>
<td>fstat</td>
<td>845</td>
</tr>
<tr>
<td>lstat</td>
<td>463</td>
</tr>
<tr>
<td>stat</td>
<td>426</td>
</tr>
<tr>
<td>access</td>
<td>204</td>
</tr>
</tbody>
</table>
tation and focusing on the usage the tests can be performed quickly, while the analysis requires more time, due to missing visualization elements.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>1</td>
</tr>
<tr>
<td>Customization</td>
<td>5</td>
</tr>
<tr>
<td>Usability</td>
<td>1</td>
</tr>
<tr>
<td>Automation</td>
<td>3</td>
</tr>
<tr>
<td>Costs</td>
<td>2</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>13</strong></td>
</tr>
</tbody>
</table>

Table 4.8: Evaluation result of the ld_preload implementation

4.7 Frida

Frida is a dynamic code instrumentation toolkit. It enables the security tester to inject self-written JavaScript snippets and libraries into application on the most common platforms, such as Windows, macOS, Linux, iOS, and Android. Frida is highly extensible by providing the tester with tools that can be used to build on top of Frida. The major development was done by Ole André V. Ravnás. He and Håvard Sørbo developed the idea in a brainstorming session about how to improve the reverse engineering process. This chapter will not cover every functionality of Frida in detail, but rather the concept and the idea behind it. Frida currently supports Android 4.2 to 6.0 with as of yet limited support for ART [44].

4.7.1 The choice of analyzing Frida

First, it is important to clarify why Frida was chosen over other dynamic analysis approaches found during the process of research. During the exploration for similar solutions like Frida, the products XPosed and CydiaSubstrate were frequently mentioned. Frida stood out since the product is constantly updated and improved, which can be seen in the Github commit history of Frida. Furthermore, the main developer Ole André V. Ravnás, holds speeches at universities and other technology conferences, which increases his credibility. For example the competitor XPosed does not provide a documentation or other knowledge base and relies on forums and third party websites. CydiaSubstrate offers better documentation, but outdated introductions from more than five years ago.

4.7.2 Architecture

Frida gives detailed insight on how it operates. On their website, the general architecture of the project is described and can be seen in figure 4.9 [44] It is not intended to describe the entire architecture of Frida in great detail, but some key components are important in order to understand how it operates. The heart of the architecture is Frida core. It injects the JavaScript, written by the tester, into the processes of the target. The injected part is referred as Frida agent in the previous diagram. This enables the tester to gain full access to the memory, hook functions and call native functions, similar to LD_PRELOAD. The communication between the tool and the injected JavaScript code is performed through a bidirectional channel.
4.7.3 Setup

The following parts are necessary for the setup of a testing environment with Frida. First of all, a host machine is required, running Linux as operating system, as well as an Android emulator using the ARM architecture. Ole André V. Ravnås announced on Github that there is at the time no active work on porting Frida to the x86 architecture. Frida can be installed on the host machine via the package managers pip and the Node Package Manager (NPM):

\[\texttt{pip install frida \# CLI tools and Python bindings} \]
\[\texttt{npm install frida \# Node.js bindings} \]

To set up frida-server, start the emulator and push the Frida server on the device, just as was done with the shared library described in the LD_PRELOAD section, and run it from the respective directory.

\[\texttt{adb push /path/to/image/frida-server-9.1.16-android-arm /path/to/image/frida-server} \]
\[\texttt{adb shell} \]
\[\texttt{cd /data/local/tmp} \]
\[\texttt{chmod 755 frida-server} \]
\[\texttt{./frida-server} \]

This leaves the server in a state to receive instructions from the client machine.

4.7.4 Experiment

According to Michael Kerrisk in his book "The Linux programming interface" [40] the four key system calls are open, write, read, and close. Therefore, those commands were issued for every individual application to log their file interaction.
sudo frida-trace -U -i write -i read -i open -i close #APPLICATION NAME
Frida-trace is a tool for dynamically tracing function calls, and just like LD_PRELOAD, it logs every open and write call that is passed from the application to the operating system. Therefore, a lot of data is generated which can be seen in the respective screenshot of each application.

**BA1** The usage of frida-trace and the BA1 functioned without any errors. An extraction of the output can be seen on figure 4.10. This dump shows several write-and-read calls that have been intercepted. Every row starts with a number of milliseconds that describes the time after the start of the execution. Whenever a system-call is logged, it is printed below the respective thread ID (TID).

A write-call expects three parameters. First, the file descriptor abbreviated with "fd" and the respective value in decimal format. Second, the buffer (buf) in hexadecimal format with the respective values what is supposed to be written. Lastly, the count parameter that defines the number of bytes written from the buffer.

Read-calls expect the exact same parameters as write calls.

**BA2** Since the BA2 could not be installed as described in chapter 4.6.5, it will not be further investigated.

```
1907 ms read(fd=0x15, buf=0xa3e6fb6c, count=0x10)
2011 ms write(fd=0x1b, buf=0xa3400000, count=0x14)
2014 ms read(fd=0x1b, buf=0xa3454c3d4, count=0x4d)
2018 ms write(fd=0x1b, buf=0xa3400000, count=0x48)
2020 ms write(fd=0x1b, buf=0xa3e6f92c, count=0x4d)
2023 ms write(fd=0x1b, buf=0xb395200c, count=0x48fe8)

2630 ms read(fd=0xsa, buf=0xbef8b18c, count=0x18)
2638 ms write(fd=0x1b, buf=0xa3400000, count=0x5c0)
2641 ms read(fd=0x1b, buf=0xa3e6fcd4, count=0x4d)
2658 ms write(fd=0x1b, buf=0xa3400000, count=0xc)
2668 ms read(fd=0x1b, buf=0xa3e6f8b8c, count=0x4d)
2677 ms write(fd=0x1b, buf=0xa3400000, count=0x10)

2895 ms write(fd=0x1d, buf=0xb3aeced5, count=0x4d)
2147 ms write(fd=0x1d, buf=0xb3aeced5, count=0x4d)

2216 ms write(fd=0x0d, buf=0xa3e6fcd5, count=0x4d)
2220 ms write(fd=0x0d, buf=0xa3e6fcd5, count=0x4d)

2262 ms write(fd=0x0d, buf=0xb3aeced5, count=0x4d)
2271 ms write(fd=0x0d, buf=0xb3aeced5, count=0x4d)

2278 ms read(fd=0x15, buf=0xa3e6fb6c, count=0x10)
2286 ms write(fd=0x0d, buf=0xb3aeced5, count=0x4d)

2301 ms write(fd=0x0d, buf=0xa3e6fcd5, count=0x4d)
2362 ms write(fd=0x12, buf=0xb5e18a0e, count=0x1)
2377 ms write(fd=0x17, buf=0xb5e18a0e, count=0x1)

2391 ms read(fd=0x15, buf=0xa3e6fb6c, count=0x10)
```

Figure 4.10: Sample output of frida-trace on the BA1

Since the BA2 could not be installed as described in chapter 4.6.5, it will not be further investigated.
BA3  Just as BA1 application, no errors occurred running frida-trace with BA3. A sample output can be seen on figure 4.11 The write calls were explained on the output of the

![Figure 4.11: Sample output of frida-trace on BA3](image)

BA1. In this extraction, open-calls are visible. The first parameter is the pathname of the file to be opened. The pathname is a char pointer. The second argument flag is an integer that defines the access rights on the file such as read-only, write-only or read-write. The function open returns the file descriptor, which is a parameter for the write and close function[40]p.70.

InsecureBankv2  An extraction of the result of tracing the InsecureBank application can be seen on figure 4.12. The output of frida-trace on InsecureBankv2 will be used to show the close-function-call. Close only expects the file descriptor of the file to be passed as a parameter. Afterwards, the file descriptor is freed.

4.7.5  Applications for Frida

The scope of vulnerabilities that are possible to be found with Frida are similar to the previously mentioned vulnerabilities presented in the LD_PRELOAD chapter 4.6.2. Another example is that Frida can be used in cases where encryption makes static analysis nearly impossible, for example when analyzing encrypted network traffic. In this case, Frida can be be used for tracing APIs, circumventing the encryption.
4.7.6 Results

Given the time frame, it was not possible to test the entire Frida framework. The documentation and video recorded talks permitted evaluating the parts that were not handled in the experiment.

Support: Frida is a well-documented and maintained dynamic code instrumentation toolkits. Tutorials and examples for different platforms, API references, and more information are available for free on their website. To contact the developers, IRC and twitter contacts are given. Frida is open source and hosted on Github, where it is possible for users to post issues and get into discussions with other users, as well as the developers. The fact that the code is open source enables the user to study and understand Frida better.

Customization: Frida is highly extensible and invites the user to use the provided API's and hack Frida. The user is encouraged to write their own scripts, and build a system that suits their needs using the provided APIs [45] [46]. Besides the APIs that Frida offers a command line interface for the sake of rapid prototyping and to simplify debugging.

Usability: In order to get started with Frida on Android, frida-server must be deployed on the device. After starting the server, one command is sufficient to get the first results,
as presented in the experiment section above. When it comes to scripting, JavaScript knowledge is required in order to use Frida to its full extent. The extensive documentation supports the usability of the entire framework.

**Automation:** Frida offers Python, C or NodeJS Bindings that can be used to automate the procedure. Depending on the effort, large parts can be automatized using the provided APIs. By default, automated testing is not possible.

**Costs:** First to mention is that Frida is open source and freely available. It is quickly usable to retrieve first results, as shown in the experiment section. The results need to be further processed which must be done by writing source code. Depending on the skill of the tester and the required functionality, the time consumption.

<table>
<thead>
<tr>
<th>Support</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customization</td>
<td>5</td>
</tr>
<tr>
<td>Usability</td>
<td>3</td>
</tr>
<tr>
<td>Automation</td>
<td>4</td>
</tr>
<tr>
<td>Costs</td>
<td>3</td>
</tr>
<tr>
<td>Sum</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.9: Evaluation results of Frida
5 Implementation

As mentioned before, the proof of concept consists of a host machine (Ubuntu 16.04) and an Android emulator. The LD_PRELOAD command expects a shared object file. These files have the files extension “.so”. In section 4.6.3, the process of compiling C applications is addressed. This shared library has the main purpose of intercepting file interaction, such as open, write, and read. It is important to note that the scope of which calls should be intercepted and logged can be extended to network connections or other system calls. The data intercepted will be sent to a Python UDP server in the JSON format and stored in a MongoDB to enable further analysis. In the current prototype, UDP was used, which can be changed to a REST API over HTTP. JSON was chosen to guarantee extensibility and enable a uniformed way of processing data. The support of receiving JSON objects could to enable the incorporation of more data from other sources, such as Frida, to increase the benefit of the program. The visualization and analysis is done in a web application that can be expanded to suit different needs, such as graphs, lists, and raw data. The current prototype simply lists the data intercepted in different ways. An overview of the implementation can be seen on figure 5.13.

5.1 Starting point

Since the entire implementation was not fully carried out during the thesis project, it is important to mention the starting point for the implementation. The basic implementation for hooking system functions and sending the raw data over UDP was given. The raw data was received by a python server and printed in the console. The system had to be modified to not only log some information and push them out over UDP, but, send specifically needed information in a well-structured JSON format. The code on the receiving Python server needed to be modified on the server side to not only print the data, but also, store them in a database. The database of choice was MongoDB since the Python programming language offers good support for it, and the JSON objects do not have to be process to be stored. A completely new component is the web application that reads out the database and presents the data, which opens up various possibilities in presenting the data gathered. In the following section, these components and their details will be explained in greater depth.

5.2 Client Side

The client side addresses the part of the implementation that is deployed on the android phone. This is the shared object, which consists of several C classes. To deploy the shared object, the Android Debug Bridge (ADB) was used, which can be used to open a terminal accessing the android system. The content of the shared library will be explained in the following chapter.
5.2.1 Structure of the shared library

Before the compilation process, the main functionality is embedded in three classes.

- libc_shim.c
- log.c
- udp_send.c

libc_shim.c contains multiple functions that will be used to override functions in other shared objects. There are hundreds of interesting functions that can be addressed, but since this would exceed the time frame of this project, the main focus was set on file interaction. An example is the function “open”, which is likely to be used in a lot of applications where any kind of file interaction is performed. The basic structure of the functions is similar, so the example of open is used to explain the basic concept of this procedure.

```c
#define WRAP(ret_type, sig) ret_type (*_nxt_##__FUNCTION__)sig =
    dlvsym(RTLD_NEXT, __FUNCTION__);
#define CALL1(p1) _nxt_##__FUNCTION__(p1)
#define CALL2(p1, p2) _nxt_##__FUNCTION__(p1, p2)
#define CALL3(p1, p2, p3) _nxt_##__FUNCTION__(p1, p2, p3)
#define CALL4(p1, p2, p3, p4) _nxt_##__FUNCTION__(p1, p2, p3, p4)

int open(const char *pathname, int flags, ...)
{
    WRAP(int, (const char *, int));
    int ret = CALL2(pathname, flags);
    char str2[300];
    snprintf(str2,300, "{%" function": "%s", ";parameters":
    ["pathname": "%s"], ["flags": "%d"], ["return":
    "%d"]}, __func__, pathname, flags, ret);
    udp_log(str2, ",", 0);
    return ret;
}
```

In the top section of the code snipped, several definitions are made. Particularly interesting is the WRAP definition, which is a macro that saves a function pointer to the original function, and is necessary to forward the function call to the original function in order to maintain the program flow. If the original function would not be called, most programs would crash, due to missing file dependencies. The different CALL definitions are mainly for passing a certain number of parameters to the original function. This depends on the number of parameters that the original function accepts. After that, a JSON-formatted string containing different variables, such as the function name, the path of the file, a flag as well as the return-value is sent via UDP. This UDP traffic will be received on the Linux host machine by the udpReceiver python class.

5.3 Server Side

The server side is a simple python script that listens on port 45500 for UDP packets. The incoming JSON objects parsed to JSON objects that can be used for further data analysis.
Therefore, this data is written to a MongoDB document database that made it possible to
directly store the JSON objects instead of parsing them for a SQL database.

```python
# Create a DB connection
client = MongoClient()

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to the port
server_address = ('', 45500)
print >> sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)
dict_filedes = {}

print >> sys.stderr, 'waiting for traffic'

def parse_json(json_obj):
    timestamp = time.time()
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S')
    json_obj.update({'timestamp': st})
    testlog = db.logs.insert_one(json_obj)

while True:
    data, address = sock.recvfrom(4096)
    try:
        parse_json(json.loads(data))
    except:
        print(sys.exc_info()[0])
```

To access the data and visualize it in a further step, the Flask Python micro framework
was used. Flask is a micro framework for Python based on Werkzeug, Jinja2 and good
tentions [47]. Flask made it possible to quickly set up a prototype web application
that reads the logs from the database and presents them on a website. The heart of this
web application is the main.py file, which acts as the view in the terms of Model View
Controller (MVC).

```python
app = Flask(__name__)
Bootstrap(app)
client = MongoClient()

log_list = []
count_fopen = 0
count_open = 0

@app.route("/")
def main():
```
log_collection = db.logs
log_list = []
filepath_set = set()

#Iterates over the database entries and fetches the necessary information
for log in db.logs.find():
    function = log["function"]

    #Stores the filepaths in a set which removes duplicates
    filepath = log['parameters'][0]['filepath']
    filepath_set.add(filepath)

    #Checks and counts how often a certain function is used (TODO
    # Make it more dynamic for other functions)
    if function == "fopen":
        global count_fopen
        count_fopen+=1
    if function == "open":
        global count_open
        count_open+=1

    log_list.append(log)

print("Number of fopen calls = "+str(count_fopen))
print("Number of open calls = "+str(count_open))
deleteAll()
return render_template('index.html',log_list=log_list,
kehrer count_fopen=count_fopen, count_open=count_open,
kehrer filepath_set=filepath_set)

#Delete all entries from the DB. Used for testing the application
def deleteAll():
    try:
        db.logs.delete_many({})
        print('\nDeletion successful\n')
    except Exception as e:
        print(str(e))
if __name__ == "__main__":
    app.run()

The main template to show the data in a browser is called index.html. It contains HTML elements, as well as Jinja2 elements. Jinja2 is a fully featured template engine for Python. It has full unicode support, an optional integrated sandboxed execution environment, widely used, and BSD licensed [47]. Jinja2 helps accessing data passed by the view and using them in the HTML code. This prototype application displays the count of certain functions, in this case fopen and open, all individual files opened by the analyzed application, as well as all calls in the original JSON format.
{% extends "bootstrap/base.html" %}
{% block title %}Android Traces{% endblock %}

{% block content %}
<h2>General statistics</h2>
<p>Number of fopen calls: {{count_fopen}}</p>
<p>Number of open calls: {{count_open}}</p>

<!-- Shows all files that have been opened at least once -->
<h2>Individual files</h2>
<ul><li><p>{{ set }}</p></li></ul>

<!-- Dump of all JSON objects -->
<h2>All JSON objects listed</h2>
<ul><li><p>{{ log }}</p></li></ul>

{% endblock %}

5.4 Visualization

Besides the currently basic data presentation, the future versions should include better ways to visualize the data. An example is that the relation of which functions were called could be displayed in a pie-chart, and critical file interactions highlighted. The JavaScript library D3 (or D3.js) allows the creation of distinct diagrams using web standards. The Stanford visualization group published an article that described the differences between D3 and other visualization tools. Commonly, tools hide the underlying scene graph, while D3 allows the user to inspect and manipulate the Document Object Model (DOM). This transparency improves expressiveness and offers a good integration with previously used developer tools [48]. The plan to use D3 for data visualization was postponed into the near future due to the limited time plan.

5.5 Traceability of API calls

Related calls can be tracked by, for example, storing the file descriptors and corresponding paths in a key-value data structure (e.g. map or dictionary) whenever an ‘open’ call is encountered. This information can then be retrieved and referenced on subsequent file
operations on the same file descriptor. When the file is closed, the corresponding entry in the map can be removed. A conceptual execution can be seen below:

```
1 = open() / fopen()
1 = read()
2 = read()
1 = write()
1 = read()
3 = read()
close() -> flush 1
```

Table 5.10: Conceptual call mapping using LD_PRELOAD

It is possible to see that one of the open functions returns the files descriptor. Afterwards, two read() calls and one write() calls are made on the same file descriptor. Since the file has not been closed yet, it can be concluded that these calls are performed on the exact same file, and are therefore mapped together. When the close() function is invoked, the file descriptor is flushed and can be reused by another open() or fopen() call. This functionality has not yet been implemented in the current proof of concept. Nevertheless, the concept on where to apply certain mechanisms have been thought through. Before storing the read() and write() function calls in the MongoDB, the file path they have operated on can be appended to the JSON object. It could still happen that a file has been opened and closed at different times. Therefore, a unique counter must be added to the JSON object in order to distinguish between these calls. This way, it is possible to simply map the calls on their file path in the web application.
6 Analysis

After the research and implementation, it is possible to answer the research questions and analyze the results. To answer research question one, a framework had to be implemented that ensured a consistent evaluation. Therefore, the metric system presented in chapter 4.2 was developed. Furthermore, the experiments had to be performed on the same applications to prevent the methods from being evaluated on applications with a different degree of complexity. The results of each approach is captured in their respective chapters and summarized in a table. The summarized results can be seen in the following table:

<table>
<thead>
<tr>
<th>Dalvik/ART</th>
<th>Native</th>
<th>Web Analyzers</th>
<th>LD_PRELOAD</th>
<th>Frida</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Customization</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Usability</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Automation</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Costs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>13</td>
<td>10</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 6.11: Comparison of evaluation results

With twenty-five being the maximum score possible to achieve, Frida is leading the field with eighteen points. In the following analysis the methods will be referred the following:

- Dalvik/ART Debugging = M1
- Native Code Debugging = M2
- Automated Web Analyzers = M3
- LD_PRELOAD = M4
- Frida = M5

**Support**: Comparing the support that is given to use one of the presented methods shows that M1 and M5 offer great support in form of documentation and tutorials while M2, M3 and M4 show room for improvement. Considering that M4 is not a product, the result is understandable. By studying the document, M3 does not require much knowledge to get the expected result. Nevertheless, there was missing documentation, or badly written documentation, that left the user in the dark regarding what is happening behind the surface of the web application. The only information that could be extracted was the result itself, or for Tracedroid from the Master Thesis report it was part of. Also, more technical support for M2 was needed to begin perform native debugging. Since the official documentation was not detailed enough, third party websites had to be consulted, which should not be the standard, especially since the provided documentation for debugging Java code was in great detail. M5’s documentation was very helpful, but lacked in-depth information.

**Customization**: Both debugging methods M1 and M2 have the same score, because their core functionality is similar. Customization is not a strong property of debuggers, since they are specialized tools used for a specific task, while other approaches are more general tools where broader functionality has a greater benefit. M3 was not customizable in any way, which makes the approach less useful due to the fact that depending on the application and case modifications in the workflow could and should occur. If the tester is interested in a certain behavior and a tool of the pool of M3 does not support this analysis, M3 becomes less useful. The current implementation of M4, as well as M5, have their
strong point in customization. Both methods can be tailored to the need of the testers to do exactly what they want it to be. They can be held minimal to fit the testers need, while scaling them to a full-size web application is possible.

**Usability:** In terms of usability, M3 stood out, since the only thing the tester needs to do is upload a file and submit the report via the provided web application. Therefore, M3 got the full five points. This extremely easy usage comes at the cost of the previously mentioned customization. M1 and M2 differ highly in their degree of usability even though they are both debuggers. Debugging Android Java code is very well implemented in the development environment Android Studio, provided by Google, which leads to the score of four points. The NDK provided by Google offers support for debugging native code on Android devices. Still, as shown in the experiment section, the setup and the actual debugging process itself was not beginner friendly. This yielded the low score of only one point for usability in the evaluation. Nevertheless, it needs be kept in mind that the author’s experience in debugging C code via the terminal was limited, while some experience in graphical debugging was present. The usage of M4 still requires a lot of manual steps in the command line in order to retrieve results. Although, if the setup is complete, the usage is simple and consists of browsing the application and reviewing the web application. The report on the website cannot yet be compared to the products seen reviewing the mentioned Automated Web Analyzers. Thus, M4 scored three points for usability. In order to get M5 set up, a few terminal commands are needed. The detailed documentation had strong influence on the usability because the testing environment was set up quickly. Nevertheless, to get started writing scripts and use M5 to its full extent the documentation must be further studied.

**Automation:** Especially interesting for the analysis of M1 and M2 is the peer reviewed journal from 2001 by Andreas Zeller, in which he discusses the topic of automating debugging [49]. The conclusion drawn by him is that automated debugging is a demanding task because of mainly two factors. One is that the program input is not the only factor, especially in modern, highly complex and interconnected systems like Android. The other is that the choice of where to set a breakpoint is the product of a chain of thoughts that is based on experience. M5 needs to be setup manually through the terminal. Still, M5 offers powerful APIs to allow the user writing script for automating processes in his tests. M4 does not offer APIs, but offers the tester complete freedom on how to design the system, and that lays the foundation for automation. Complete automation is by now out of the scope, since dynamic analysis is a highly reactive task. Applications have different attack vectors and functionalities that require the process of realistic user input. An example is that in order to automate a dynamic test on a banking application, a user account is needed in order to fully use the application. Furthermore, the automated system would have to insert correct credentials and perform transactions, which is difficult to automate. The tool ApkScan, which belongs to one of the tools of M3, produced random input, which is a starting point, albeit insufficient.

**Costs** The costs to perform dynamic analysis consist mainly of the costs of work hours, because every approach was feasible to do without commercial products. As shown in the results, commercial tools exist, but could not be used. The time used for performing dynamic analyses relies heavily on the expertise and pricing of the tester. As shown in the results of M1 and M2, debugging a full-scale application can be tedious and slow. M1 scored only one point in costs, since, in case of a black-box test, retrieving the source code was not possible in a reasonable time frame. Using tools from M3 took only a few minutes. Still, one of the most promising tools, namely App-Ray, was fully commercial, while ApkScan was limited in the free version. Furthermore, the analysis of the report
varied for different products. Due to the fact that Tracedroid had several text files to inspect, this takes time for the analysis. For these reasons, the score was reduced to three points. For M4 and M5, the costs heavily depend on the scope of the self-implementation. First, results can be achieved quickly and in less time, while further functionality requires costs of implementation. Depending on the aim of the system, as well as the experience of the tester, the costs require varying amounts of working hours.

### 6.1 Comparison of metrics

Besides the results of the dynamic analysis methods, this chapter aims to analyze the score of the metrics itself. The results can be seen in the table below.

<table>
<thead>
<tr>
<th>Sum</th>
<th>Support</th>
<th>Customization</th>
<th>Usability</th>
<th>Automation</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>16</td>
<td>14</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.12: Comparison of the metric scores

With a score of sixteen points, the highest amount of points throughout different dynamic analysis methods was given to usability. Nevertheless, the maximum of twenty-five points shows that there is a lot of room to improve the usability, which would automatically lead to a time/cost improvement. Automation and customization share fourteen points, which is just above the 50% border with 12.5 points. As previously mentioned, automation a process that needs to be more closely investigated at when talking about large-scaled tests, but also for small-scale tests, efficiency can be increased by automating steps. Furthermore, the previously mentioned problems with automating dynamic analysis will slow down the development for products like this. Being below 50%, the support of the solutions is not sufficient. Be it the lack of good documentation, or no sufficient way to get in contact with the developers, there are weaknesses that need improvement. The factor cost is difficult to analyze, because it depends on several factors. Nevertheless, the score of eleven points can be considered realistic with regards to the fact that dynamic analysis still requires a lot of manual work and can be tedious.
7 Discussion

In the discussion we will go through the entire report to discuss every section's outcome, and relate to the research questions set up in section 1.5. It will act as a summary and relates to previous work in the area of software security testing.

7.1 Theoretical context

The theoretical context chapter is giving the reader, if not present yet, the necessary frame to understand the content of later chapters. For the readers who are not familiar with software testing, it is imperative to explain the term "dynamic analysis" since it is mentioned in the title of this thesis work. Mentioning the different approaches developing software for Android is important as well to give the reader an understanding that dependent on the development technology the structure of the APK will differ. Without this knowledge, the reader would not know that there are applications that have been developed with a different programming language as a core than Java. The Android system is a relatively modern operating system and provides the presented security features. Security concepts of the Android operating system, such as sandboxing, or the implementation of secure interprocess communication have been presented. Software developers should not dive directly into the development process, but study the basic security mechanism existing in order to develop more secure applications. There exists a need for secure applications, as shown by the report of Paladion Mobile Security Team mentioned in the background section. The main resources used in section 3 are from the official Android documentation provided by Google. This was only possible because Android was designed to be open-source from the ground up.

7.2 Dynamic analysis methods

Evaluating different analysis methods is the core of this thesis work. The creation of a frame for evaluation was the starting point and offered different possibilities. Defining the metrics is essential for achieving a comparability between the methods. In another thesis, dynamic analysis methods could be compared on completely different metrics. In the article "Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation", mentioned in section 1.2 the authors focused on obfuscation, evasion of analysis, and the evaluation of decompilers which were different metrics the tools were compared on than in this work. The choice of methods to investigate is another decision that could have steered the focus on another level. Instead of focusing on the in-depth technical functionality the evaluation shows the area of use and details about the general usage of the tools. During the research phase, multiple ideas on how to extend the scope of the thesis came up. An example could be to find an application with a known vulnerability and create use cases to show how good a tool is in finding the specific flow. The results would point out the functional strengths of certain methods. Nevertheless, the results are especially useful for people who want to start with Android software testing and need some orientation on what possibilities exist. For professional testers who have their toolset and knowledge, the results can be and update but might also be redundant to their experience.
7.3 Traceability and Visualization

The traceability and visualization of system calls is a feature that has not been found in any tool during the research phase and is therefore desired to implement in the proof of concept. Due to the fact that more time was needed for evaluating the dynamic analysis methods, the implementation was started but not finished. Mapping API calls for file access is currently present in a conceptual form, which shows the idea on how the mapping will be implemented. Similar to mapping file access the same concept can be adapted in order to map access to the same network connections. In the article "API Tracing Tool for Android - Based Mobile Devices" [50] the authors introduce a prototype application that utilizes the ADB instead of manual LD_PRELOAD. They use the built-in profiling functionality of the ADB. The execution of the process will be monitored which includes method invocation and resource utilization with the respective timestamps. The tool presented wraps the built-in profiling functionality from Android, which is on the Java level, while the tool created in this thesis is a completely new approach and operates on the system level. The advantage is, that no system call gets performed, while the approach on the Java level cannot provide this coverage.

7.4 Application of different dynamic analysis methods

The results of research question three show that all solutions besides the automated analyzers are rather specialized. LD_PRELOAD and Frida are aiming to intercept all communication closest to the operating system, while debuggers operate more specific and require source-code to be used. There are several use-cases for different tools. Dependent on the ability to customize a tool, the capabilities on what vulnerabilities, bugs, and other flaws can be detected vary greatly. There is not a single tool that could make all other tools obsolete; the skill to choose the right tool for the right problem is a skill an experienced software security tester should develop.
8 Conclusion

The thesis project yields results in an extensive evaluation of dynamic analysis methods. The evaluation is interesting for people working in the IT sector to extend knowledge about software testing and approaches for performing dynamic analysis. For software security testers the evaluation of today’s dynamic analysis methods is an update and can shed new light into the current workflow. Another result is the proof of concept, which is the foundation for developing a dynamic analysis product with LD_PRELOAD as core. Furthermore concepts for the traceability and visualization were presented, and will be implemented in near future. The evaluation of the thesis turned out to be broader than expected, and took more time so that the implementation of the API tracing and visualization had to be postponed. Considering every possible factor that affects a dynamic analysis test is a skill, that cannot be learned in the scope of a Bachelor thesis. More experience in the area of software security testing could have yielded different and probably more detailed results.

8.1 Future research

The presented prototype is the foundation of a tool that will be developed in cooperation with TrueSec. An important step is to improve the visualization of all data received from the Android device. The concept of machine learning in dynamic analysis is a topic addressed in several papers found during the research phase. Categorizing errors and building a database can be used to detect potential vulnerabilities, improving the dynamic analysis workflow in the presented implementation, but also for other approaches. Yurii Brun investigated this in his master thesis with the title *Software Fault Identification via Dynamic Analysis and Machine Learning* [51].
References


mated Android App Security Assessment.”


[38] A. Allain, “Shared libraries with GCC on Linux.” [Online]. Available: 

ndk/guides/android[_]mk.html


//www.tutorialspoint.com/c_standard_library/c_function_fread.htm

tutorialspoint.com/c_standard_library/c_function_fwrite.htm

//man.openbsd.org/ld.so.1

//www.frida.re/docs/hacking/

javascript-api/


Available: http://flask.pocoo.org/

http://vis.stanford.edu/papers/d3

partnerID=40&md5=070a7a1d0f026aa56a9166a9c8515c6b

[50] S. Choi, M. Bijou, K. Sun, and E. Jung, “API Tracing Tool for Android-Based Mo-
obile Devices,” vol. 5, no. 6, 2015.