

DAMN - a Debugging Tool for Source Code Reverse Engineering and Dynamic Manipulation Live on Android Devices

GERALD SCHOIBER



MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Mobile Computing Master

in Hagenberg

im Januar 2016

© Copyright 2016 Gerald Schoiber

This work is published under the conditions of the *Creative Commons License Attribution–NonCommercial–NoDerivatives* (CC BY-NC-ND)—see <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, January 11, 2016

Gerald Schoiber

Contents

Declaration	iii
Preface	ix
Abstract	x
Kurzfassung	xi
1 Introduction	1
2 Related Work	3
2.1 TaintDroid	3
2.2 AppFence	3
2.3 DroidScope	4
2.4 DroidTrace	4
2.5 API Monitor & Aurasium	4
2.6 Mobile-Sandbox	4
2.7 ANANAS	4
2.8 ANDRUBIS	5
2.9 Google Bouncer	5
2.10 Summary	5
3 Android	7
3.1 Versions	7
3.2 Distribution	8
3.3 Java	9
3.3.1 Reflection	9
3.3.2 JNI	9
3.4 Linux	9
3.5 Architecture	10
3.5.1 Kernel	10
3.5.2 Init	11
3.5.3 Runtime	11
3.5.4 System Services	12

3.5.5	Applications	13
3.5.6	Application Permissions	13
3.6	Access Control on Android	14
3.6.1	Discretionary Access Control	15
3.6.2	Mandatory Access Control	15
3.6.3	SELinux	15
3.7	IPC	15
3.7.1	Sockets	16
3.7.2	Named Pipes	16
3.7.3	Binder	17
3.8	Boot Process	17
3.9	Zygote	18
3.10	Android Tools	18
3.10.1	ADB	18
3.10.2	NDK-Build	19
4	Reverse Engineering	20
4.1	General Reverse Engineering Term	20
4.2	Disassembler	21
4.2.1	Baksmali	21
4.2.2	Apktool	21
4.3	Decompiler	21
4.3.1	JD-Core	22
4.3.2	JAD	22
4.3.3	Android Decompiler	22
4.4	Analyzing Reversed Source Code	23
4.5	Obfuscation	26
4.5.1	How it Works	26
4.5.2	Summary Obfuscation Techniques	27
4.6	Obfuscation Tools	27
4.6.1	ProGuard	27
4.6.2	DashO	28
4.6.3	Decompiled Obfuscated Source Code	28
4.7	Used Decompiler in DAMN	32
4.8	Limitations	32
4.9	Analysis Methods	32
4.9.1	Static Analysis	32
4.9.2	Dynamic Analysis	33
5	Concept	34
5.1	State of the Art	34
5.2	Goals	34
5.3	Scope of DAMN	35
5.4	Use Cases	35

5.4.1	Security Researcher	36
5.4.2	Software Development Company	36
5.4.3	Malicious Attacks	36
5.5	Summary	36
6	Tooling	37
6.1	SuperSU	37
6.1.1	Installation	37
6.1.2	Installation Process in Detail	37
6.1.3	Usage	38
6.2	Dynamic Manipulation	38
6.2.1	Cydia Substrate	39
6.2.2	Xposed Framework	39
6.3	Xposed	41
6.3.1	XC_MethodHook Class	42
6.4	Jadx	42
6.5	Civetweb	42
6.5.1	WSS	43
7	DAMN	44
7.1	Overview	44
7.2	Architecture	44
7.2.1	Application Layer	46
7.2.2	Runtime Layer	46
7.2.3	Native Library Layer	46
7.3	Interaction Structure	47
7.3.1	File	47
7.3.2	IPC	47
7.3.3	WSS	48
7.3.4	Communication Trace	48
7.4	DAMN User Interface	49
7.4.1	DAMN Application Activities	49
7.4.2	DAMN Browser Pages	50
7.4.3	Start Page	51
7.4.4	Tracking Page	53
7.5	Configuration File	54
7.6	Flow of Loading a Tracked Application	55
7.7	DAMN Server Process	56
7.7.1	Communication	57
7.7.2	Document Directory	57
7.8	DAMN Xposed Module	57
7.8.1	Hook Process	58
7.8.2	Control Flow Architecture	58
7.9	DAMN Runtime States	58

7.9.1	Run	58
7.9.2	Pause	58
7.9.3	Step	59
7.9.4	States of Investigated Application	59
7.9.5	Obfuscated Applications	60
7.10	Manipulate the Application	60
7.10.1	Manipulation of Parameters	60
7.10.2	Manipulation of Return Value	61
7.10.3	Manipulatable Classes	62
7.11	Behavior Rules	62
7.11.1	Structure	62
7.11.2	Triggers	63
7.11.3	Actions	63
7.11.4	Chaining Triggers or Actions	64
7.11.5	Current State	64
7.12	Web Socket Data Exchange Protocol	64
7.12.1	Protocol Structure	64
7.12.2	Protocol Codes for Start Page	65
7.12.3	Protocol Codes for Tracking Page	65
7.13	Summary	66
8	Investigating Real World Applications	67
8.1	Test Environment	67
8.1.1	Hardware	67
8.1.2	Software Used on Device	67
8.1.3	Software Used on Computer	68
8.1.4	Used Tools	68
8.2	Setup	68
8.3	Simple System Application	70
8.4	Third Party Applications	74
8.4.1	Quiz Application A	75
8.4.2	Investigate Quiz A	76
8.4.3	Quiz Application B	80
8.5	Recap	83
9	Future Work	84
9.1	Behavior Rules	84
9.2	USB Tethering	84
9.3	Multi Threading	85
9.4	Stability	85
9.5	Outlook	85
10	Conclusion	86

Contents	viii
A Content of CD-ROM	87
A.1 PDF-Files	87
A.2 Others-Files	87
A.3 Image-Files	87
A.4 Implementation-Files	88
List of terms	89
References	91
Literature	91
Online sources	93

Preface

I would like to express my very great appreciation to Univ.-Prof. PD DI Dr. René Mayrhofer for his valuable and constructive suggestions during the development and implementation of my thesis. I would also thank the staff of the following organizations where I worked and shared ideas from various projects:

- University of Applied Sciences Upper Austria
- Josef Ressel Center u'smile
- Institute of Networks and Security

My grateful thanks also extends to my girlfriend Anna who was patiently proofread this thesis and gives me motivation to proceed.

Abstract

Attackers use reverse engineering techniques to gain information which can be used to manipulate applications for their purpose. The knowledge about the reversed information are not restricted anymore to such attackers because DAMN provides an easy to use tool for reversing Android applications. This can be used to build applications more secure and makes it harder for attackers to touch them.

DAMN can handle obfuscated source code as well. As it combines reversed source code and dynamic manipulation techniques it can provide a new way to investigate obfuscated source manually. It is possible to hook into a running application and stop it at any given time. Furthermore it can manipulate values which are passed through method calls and give the opportunity to test against various constellations.

Kurzfassung

Angreifer benutzen Reverse Engineering Methoden um Information aus Anwendungen zu bekommen und sie für ihre Zwecke zu nutzen. Das Wissen über die Informationen, die aus diesem Prozess gewonnen werden können, sind nicht nur mehr auf Angreifer beschränkt, da DAMN ein leicht zu verwendendes Werkzeug bereit stellt, Android Anwendungen zu untersuchen. Diese Information kann benützt werden um sichere Anwendungen zu schreiben, welche es für Angreifer erschwert, diese anzugreifen.

DAMN kann auch mit verschleierte Quellcode umgehen. Da es Quellcode aus Reverse Engineering Methoden und dynamische Manipulationstechniken verbindet, bietet es einen neuen Weg verschleierte Code manuell zu untersuchen. Eine ausgeführte Anwendung zu unterlaufen und jederzeit stoppen zu können bietet neue Möglichkeiten. Es sind auch Änderungen von Parameterwerten möglich, die bei Aufrufen von Methoden übergeben werden und somit können auch alternative Konstellationen getestet werden.

Chapter 1

Introduction

The smart phone became a central object in our daily life. People uses it to take pictures, message, post and phone with other people. The circumstance that their device is always in range and the huge amount of private data which is stored on it makes it to a very critical item. Security is therefore mandatory to ensure privacy.

For this purpose, we need to have the possibility to analyze applications which we use to get certain that they are safe. To make it possible on Android where we do not have the source, it is common to reverse the application to get the source code out of it. Due to the fact that Android uses Java for the application environment, reversing such applications to get readable source code is pretty straight forward. Especially on applications where they do not use any obfuscation. As obfuscation only shifts the problem of reversing by adding some more complexity to the source, it does not change anything in conceptional security flaws. From this point of view, we need a powerful tool which can handle obfuscated source code in a handy way.

While there are some tools out there for reversing, none of them are conceptional handle obfuscated source code which results often in long and very hard investigations and analysis to understand the source. This is where DAMN comes in. It makes it possible for developers, researchers or even advanced users to analyze applications on their own device. After installation and configuration, the user can start any application on the device and analyze step by step every method call from the very beginning till the end. It makes it possible to look at the source code, as well as the passed parameters and their values. The same applies on the return value and any fields. Even more, it gives the possibility to change those values on runtime to test the behavior directly.

The structure of this thesis has the intent to first inform shortly about related work. After this we take a look into Android and some of the Linux components which are used by DAMN. Furthermore we have a look at the reverse engineering process and how we can use it. Combined with an overview

about tools and projects are used in DAMN we describe how our tool is working and how it can be used to investigate applications. In addition to that we will investigate third party applications of Android and show the results. Lastly we give an outlook of the future work and lessons we learned during the implementation in the conclusion.

As previously mentioned, this thesis and tool were built to make it possible to analyze applications on Android and make security analysis easier. The fact that this tool perform black box reversing and gives the possibility to manipulate applications makes it also a security issue itself. Arbitrary usage can potentially lead to more malware which make use of such issues.

On the other hand it also provides software developers with the opportunity to take a closer look into their own applications from the view point of an attacker. Then this information can be used to secure the application and make it more secure which brings benefits for developers as well as users.

The way this tool gets used is up to the users and can help both parties equally. In the end it will lead to security improvements and make Android applications less attackable as today.

Chapter 2

Related Work

Before we introduce DAMN we are looking at the related work. Since our tool is intended to be used for application analysis we take a look which other tools performs application analysis. Almost all analysis tools in our related work are full or semi automated analysis tools for detecting malware. Some of them are using emulators for dynamic analysis because they do not need different hardware if they want to test against different Android versions. Others have to use special kernels or have to manipulate the application on byte code level to analyze them. Lets have a look at the different solutions and how they are performing.

2.1 TaintDroid

Enck et al. [10] propose a system called *TaintDroid* who tracks the flow of privacy sensitive data through applications in realtime. This is done by labeling/tagging such sensitive data with a taint and track whenever an application access it. If this happens, TaintDroid notifies the user(and also other applications) about it. A study of them shows that about two-thirds of 30 popular Android applications leaking user sensitive content. Unfortunately it does not block such unwanted data leaks.

2.2 AppFence

A solution to protect user against such data leaks is provided by Hornyack et al. [13] with the tool *AppFence*. It uses TaintDroids tainting methods for their implementation. They implement two approaches to protect the data. The first is shadowing sensitive data, which means that they will return some fake data if an application requests it(e.g. location data). The second approach is exfiltration blocking. This means, whenever AppFence detects tainted data is written to a socket, it will drop the data and either fake a send conformation or tell the application that the device is in airplane mode.

This is a very lean way to protect user against collecting sensitive data of them.

2.3 DroidScope

DroidScope, which was written by Yan and Yin [27] does not run on real devices as the solutions above. Instead, it runs on the Android emulator and gives the opportunity to analyze native code components. However, it was built to detect malware and does not implement any manipulation capabilities.

2.4 DroidTrace

The only solution that provides forward execution is *DroidTrace* developed by Zheng et al. [28]. It is a dynamic analysis tool based on *ptrace*(process trace [18]). They disassemble the application into *smali code*(named after this project¹), analyze it and create a function flow graph. With changes on smali code, which then gets repacked afterwards, they are able to trigger different dynamic loading behaviors for investigation purpose. *DroidTrace* is able to find certain zero-day malware during its analysis on a larger scale.

2.5 API Monitor & Aurasium

The developers of *API Monitor & Aurasium* Xu et al. [26] take another approach. Since they are only repacking the application with the *apktool* 4.2.2 and add some additional code into the package, they do not take use of privilege access. If a violation occurs, the user will be ask if this violation will get accepted or deny it.

2.6 Mobile-Sandbox

Another solution is *Mobile-sandbox* which combines dynamic and static analysis for automatic testing. It also logs calls to native *APIs*(Application Programming Interface). More information about this project can be found here [21].

2.7 ANANAS

ANANAS is an expendable automated static and dynamic malware analysis framework for analyzing Android applications. It allows to write plugins to

¹<https://github.com/JesusFreke/smali>

extend the functionality and change settings. The framework raises events where those plugins can react to them and execute additional code. Logs will be saved into a database where filters allows efficient reports. For executing applications, ANANAS uses the emulator which is shipped with the Android *SDK* ². To interact with the emulator, it provides a scripting language which can simulate interaction with the application as user input, battery status change, incoming call simulation and more [6].

2.8 ANDRUBIS

ANDRUBIS is another automated static and dynamic malware analysis tool which run in a *QEMU* (a generic open source machine emulator and virtualizer) runtime environment [25]. It analyzes the manifest file with a static analyze method as well as the actual byte code. The dynamic analyzing part monitors the *Dalvik* 3.5.3 virtual machine and also the system level. After investigation it performs additional analysis such as network traffic. The analysis methods including tainting sensitive data allow to trace, if those data get touched by an application. *ANDRUBIS* analyzed one million applications from the *Play Store* and published the results in this paper [14].

2.9 Google Bouncer

Google Bouncer is an automated dynamic analyzing tool for Android applications. It is not exactly known how it works because Google keeps it secrete. But it seams that it uses *QEMU* [1] to simulate an Android device. *Jon Oberheide & Charlie Miller* were trying to get some more information published here [30]. Google implemented it as a service which analyze every application on the Google Play Store [5]. If they detect any malicious behavior they will remove it from the store. Bouncer is not a perfect analysis tool as *Nichoas Percocos* researches shows on *Black Hat*³ in 2012 [19] and it never will because this way of analyzing is very complex.

2.10 Summary

Most of the tools we introduced are automated analyzing tools and they are analyzing mostly both ways, static and dynamic. Static analyzing can only hardly detect dynamic code execution and struggles with the completeness of the decompiled source code. Dynamic analysis overcome this problem because they are not needed for that purpose. Since dynamic analysis are very complex, automated software can hardly handle all possibilities and making

²<http://developer.android.com/sdk/index.html>

³<https://www.blackhat.com/>

further investigations is necessary. For those manual analysis we can use various decompilers which can reverse source code. Because obfuscated source code is hard to investigate manually, those investigations have additional overhead.

Dynamic reversing is rarely used for manual investigations or have a high efforts on setup before it can be used. We would not have any problems with obfuscated code, because the program have to be still runnable regardless of whether obfuscation is used or not. Of course, we could use a disassembler which offers possibility to make changes on the code and reassemble it to an application, but assembler code is harder to read than well known Java code.

DAMN provides a solution for that problem and combines the advantages of readable Java source code and dynamic debugging of applications. As an additional feature, it also brings the possibility to manipulate the investigated application and test the behavior on different circumstances.

Chapter 3

Android

Android is a Linux-based operating system for mobile devices. Over the last years it becomes one of the most used mobile operating systems worldwide [29]. We have a look at some basic concepts of Linux as base for our investigations on Android and of course for DAMN afterwards. First we want to get some further informations about the components and features of a general Linux system and after that we are looking at some Android specific features. This should represent a short introduction into Android to get a better understanding how DAMN is interacting with the system and is able to perform its features.

3.1 Versions

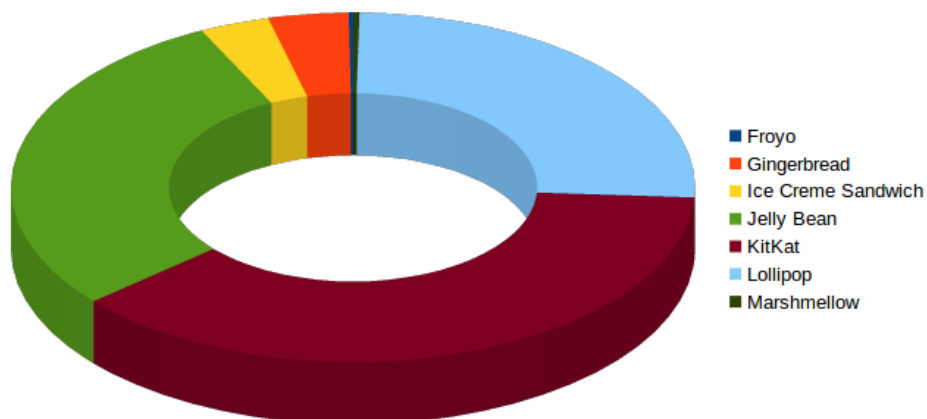
Since Android started in September 2008 with the version 1.0, it was developed fast and a new version was released every few month. In table 3.1 we are giving an overview about all versions. On version 1.5, Android started to name their releases names of sweets which we also put in this figure. One special release was in June 2014. Android released the Android *Wear* which gives support for wearables like smart watches.

Version	API	Release Date
1.0	1	September 2008
1.1	2	February 2009
1.5 Cupcake	3	April 2009
1.6 Donut	4	September 2009
2.0 - 2.1 Eclair	5 - 7	October 2009
2.2 - 2.2.2 Froyo	8	Mai 2010
2.3 - 2.3.7 Gingerbread	9 - 10	December 2010
3.0 - 3.2.1 Honeycomb	11 - 13	February 2011
4.0 - 4.0.4 Ice Cream Sandwich	14 - 15	October 2011
4.1 - 4.3.1 Jelly Bean	16 - 18	June 2012
4.4 - 4.4.4 KitKat	19	October 2013
4.4W	20	June 2014
5.0 - 5.1.1 Lollipop	21 - 22	November 2014
6.0 - 6.0.1 Marshmallow	23	October 2015

3.2 Distribution

Although the actual version of Android is *Marshmallow*, the distribution of Android versions shows the fragmentation this operating system is facing 3.1. *KitKat* and *Jellybean* take a big peace of the cake and a lot of versions which are actively used have known security flaws¹.

Figure 3.1: Android Distribution



¹https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html

The actual list of distribution can be found on Androids developer page which is also updated frequently².

3.3 Java

Java is an object-orientated program language which was developed with high portability in mind. To achieve this, Java compiles its applications into *byte code* that can be interpreted by a virtual machine, the *Java Virtual Machine*(*Java Virtual Machine*). The virtual machine on Android is the *Dalvik 3.5.3* virtual machine where the compiled application runs on. On devices where ART is used, the applications will be additionally compiled into device specific code before it is running on the ART runtime.

Nowadays Java is a very common used program language which is a good base to get a lot of developers to write applications for the Android operating system³. The fact that it is object-orientated makes it easier to structure an application and also reuse code parts [22]. Another interesting feature is *Reflection* which will be describe in the next section.

3.3.1 Reflection

The feature which makes it possible to inspect classes and objects on runtime is called *Reflection*. It also allows to manipulate those objects. It provides methods where we can investigate how a class is structured and get fields and methods of it. Furthermore it is also possible to dynamically create instances of classes and call methods of it [22].

3.3.2 JNI

Java provides also a feature to interact with native code that is called *JNI*(Java Native Interface). It can load shared libraries which can be used over this special interface. In some cases it might be useful to write specific code parts in native code(e.g. C or C++) as it can bring performance benefits. As native code do have other types of variables as Java, JNI provides also a way to convert this types [22]. On Android provides also additional information in the developer page⁴.

3.4 Linux

Before Linux there was *Unix*. It was created by Dennis Ritchie and Ken Thompson in 1969. The simplicity of its design and the available source code

²<http://developer.android.com/about/dashboards/index.html>

³<http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>

⁴<http://developer.android.com/training/articles/perf-jni.html>

lay the foundation to develop multiple Unix systems which benefits from each other. Thus made the operating system powerful, robust and stable. As time passed by, some companies made their own commercial product that leads to license changes. So customers were not allowed to make changes on the source code anymore.

This is where Linux comes into play. Linux wanted a free and open operating system where it is possible to everybody to make changes to the source. The first release was in the late 1991. Since then a huge community of developers arose and Linux became a very important operating system which is based on the Unix design [15].

3.5 Architecture

After this short excursion into Linux history we will look closer into the Android system and its Linux related features. Android has made some changes into the kernel itself, so it is not a pure Linux kernel anymore. In this section we shortly look into the Android environment and how the system is organized. This is important to understand how our tool is implemented and communicates in the system. The figure 3.2 shows the general architecture of Android.

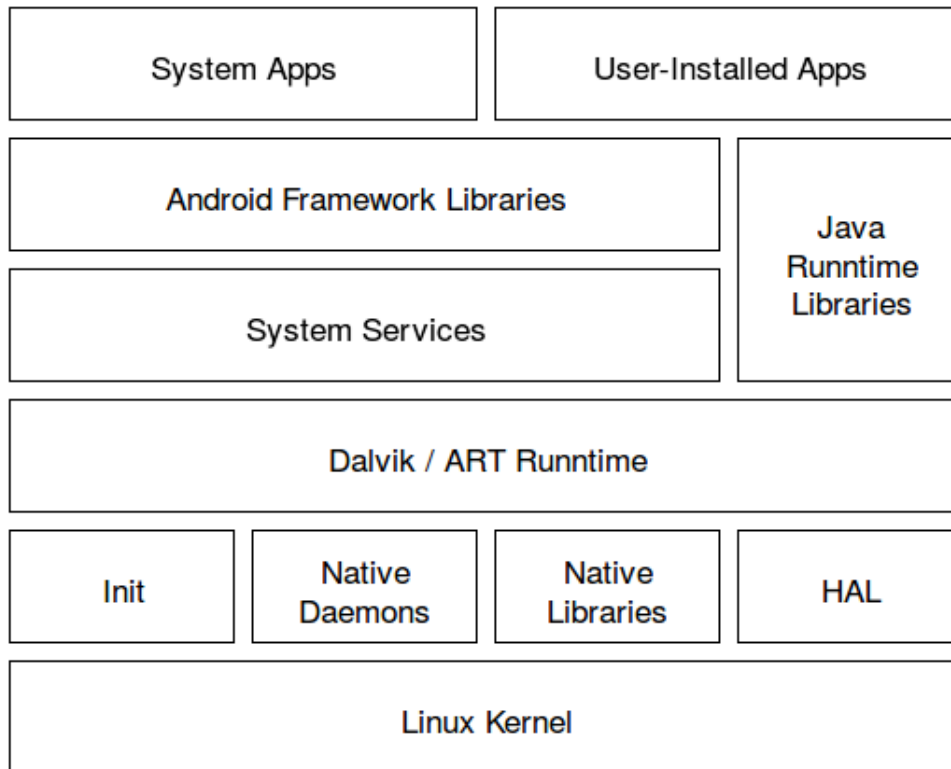
We shortly describe the different parts in this figure at the next few sections.

3.5.1 Kernel

The *kernel* is the base of every Linux operating system. The same applies to Android. Android made some changes to the kernel therefore it is not possible to use a basic Linux kernel. There are attempts to run a *vanilla kernel* on Android but this is not the usual base of an Android operating system. A vanilla kernel is a standard Linux kernel which does not have any additional functionalities as kernels which are provided by distributions. However, the kernel provides different mechanisms for integrity of processes as well as hardware support. Here are some of the additional features of the Android kernel:

- Binder
- Ashmem
- Logger
- Wakelock
- Alarm Timers
- Paranoid Network Security
- Timed Output & Timed GPIO

Figure 3.2: Android Architecture



There are more features which are listed here⁵.

3.5.2 Init

The *init* process is the first that gets started in the *user space*. User space is the term which is used when processes do not run in the kernel. This happens directly after the boot process 3.8. In differ to the Linux *init* process, the Android *init* was built from scratch as well as the *init* scripts. The *init* process starts all other userspace programs, regardless of whether it is a system process like *adbd* 3.10.1 or the *Zygote* 3.9. Later we will take a deeper look into this daemon because our tool needs to get started from this point.

3.5.3 Runtime

Android applications are basically written in Java, but they could also use parts in native code such as *C* and *C++*. In differ to normal Java applications which run on JVM, Android runs them in an own runtime environment

⁵http://elinux.org/Android_Kernel_Features

called either Dalvik or ART. Both are acting as a sandbox which protects applications to interfering each other on the system.

Dalvik

Dalvik was the first runtime on Android. It was designed with mobile devices in mind to use as less resources as possible. One of the biggest differences between Java and Dalvik virtual machine is that the JVM is stack-based while Dalvik has a register-based architecture. This also leads to different instruction sets which can be used. In general, register-based machines use fewer instruction to achieve the same as stack-based machines which makes it ideal for mobile usages [20].

Dalvik cannot execute the Java class files directly, instead it uses *dex* files. There is also another format which can be executed called *dex* file. Odex stands for *Optimized Dalvik Executable* which is, as the name suggests, an optimized Dalvik executable file. This special format splits some parts of an application into such a odexed file which can be directly loaded and do not have to be extracted from the apk file⁶.

ART

With the Android *L (Lollipop)*, Android got a new runtime called *ART*. On Android *L* it was still possible to switch back to Dalvik but it is standard on Android *M* alias *Marshmallow*. As *M* was still under development during the creation of DAMN we did not care about it. Fortunately DAMN is compatible with it as we tested it on a *Nexus 6* with *M* on it. Because DAMN relies on Xposed, and this tool has already adopt changes to make it work on that new runtime.

A big difference to Dalvik is that ART is implemented as *AOT(ahead-of-time)* runtime. This means that ART compiles the applications on installation into a device specific executable file which should bring execution performance improvements. The compiled files have the *oat* file format which can be directly loaded into the memory and are comparable in they functionality with odexed files⁷.

3.5.4 System Services

System services are executed on top of the runtime and provide Android features to other applications. We can get a list of all available services on a device with the command *adb shell service list*. Depending on the version your device is running it will print a list about one hundred services. Here are some of them listed:

⁶<https://www.androidpit.de/root-custom-rom-unterschied-odexed-und-deodexed>

⁷<https://source.android.com/devices/tech/dalvik/>

- sip
- phone
- nfc
- simphonebook
- telecom
- fingerprint
- backup
- usb
- audio
- wallpaper
- search
- country_detector
- location
- notification
- alarm
- activity
- user

As we can see the services have a broad range of functionalities they provide. Almost every of this services can be used by an application or service over a defined remote interface if they have the right permission to do so [8].

3.5.5 Applications

While Linux is building the base for Android, applications give the possibility to extend functionality to the device to the user. Android is shipped with some pre-installed applications which providing some basic functionalities like the dialer or the camera application. Every application is running in its own installation environment and has its own permissions to interact with other applications or features of Android. The installation directory differs and can be programmatically experienced like this:

Listing 3.1: Get Installation Directory

```
1 getApplicationContext().getFilesDir().getAbsolutePath();
```

This will give us something like `/data/data/<app package>/` which DAMN needs to put some additional data there which is described in more detail later.

3.5.6 Application Permissions

Applications need to be restricted in the way of accessing sources of the device. E.g. we do not want a simple game to read all of our SMS messages

we received. Therefore the system has to provide the possibility to restrict such access. In Android, user can see what application needs which permissions and can decide if it is plausible or not. Unfortunately the Android ecosystem only provides us the opportunity to either allow all permissions an application claims or to not to install it.

The permission model in Android relies among others on the Unix *DAC* 3.6.1 mechanism. If an application has the permission for using the Internet, it will be added into the group with the *gid inet*. So to take a look at the permission mappings on the device we can look into the */system/etc/permissions/platform.xml* [3]:

Listing 3.2: Snipped of Permission File

```
1 <permission name="android.permission.INTERNET" >
2   <group gid="inet" />
3 </permission>
4
5 <permission name="android.permission.READ_LOGS" >
6   <group gid="log" />
7 </permission>
8
9 <permission name="android.permission.READ_EXTERNAL_STORAGE" >
10  <group gid="sdcard_r" />
11 </permission>
12
13 <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
14  <group gid="sdcard_r" />
15  <group gid="sdcard_rw" />
16  <group gid="media_rw" />
17 </permission>
```

We will later look deeper into the access permission model of Linux that are partly used to implement those permissions of Android applications.

On Android *M* it is now possible to decline single permissions for each application. Additional information about this can be found on the Android developer page⁸.

3.6 Access Control on Android

Android uses also access control mechanisms from Linux. These are important functionalities to ensure that every process can only access resources where it has the permissions to it. Some of the application permissions also relies on this access control mechanisms.

⁸<http://developer.android.com/training/permissions/index.html>

3.6.1 Discretionary Access Control

A very basic access control mechanism is *DAC*. It is used for the Linux file systems and handles the access control on the system. The concept can handle different users as well as groups. A non privileged user can change the permission to files which he owns and make it e.g. readable for every other non privileged user on the system. This can be done without an administrator which is handy in many cases but can lead to inconsistency.

3.6.2 Mandatory Access Control

Another access control mechanism is *MAC*. The very basic difference between DAC and MAC is that a system administrator must define policies which declares which user or group has access to which system resources. They can not be changed by a non privileged user. An example of MAC is *SELinux* which is described below.

3.6.3 SELinux

As explained above, *SELinux* is a mandatory access control system which extends the standard Linux DAC mechanism. The problem with DAC mechanisms is that any application could access files which are public read or writable regardless of whether it was intended or accidentally.

Android uses a modified implementation of SELinux that is implemented since version 4.3⁹. The purpose is to split core systems into security domains. For every domain there are existing access policies which control the accessibility between them. These policies can not be changed by a common application and therefore are fixed. On Android version 4.4, SELinux is in enforcing mode which applies to the core system daemons. Applications still run in permissive mode and that is why violations of applications only produce logs but no runtime errors [8].

3.7 IPC

In Android, the kernel takes care of each process is running in its own separated address space. Therefore a process can not simply manipulate the memory of another process. This provides stability as well as security to the system. But if a process wants to use a service, which actually is just another process, we need a way to interact between the processes. For that purpose Android and any other Linux based systems use *IPC* (Inter Process Communication). There are a lot of different opportunities to achieve such a communication between processes like sockets, signals, pipes, semaphores

⁹<http://seandroid.bitbucket.org/>

as well as the *Binder* 3.7.3. Let us take a brief look at some of those mechanisms.

3.7.1 Sockets

Sockets are a great way to interact with other processes or even other systems over a network from an application. To create such a socket we use a system call named *socket* [5]. This call need three parameters:

- Domain
- Type
- Protocol

The *domain* determines which protocol family is used for communication. To get a list of possible protocols we can use the command `cat /proc/net/protocols`. The following list shows some domains:

- PF_UNIX
- PF_INET
- PF_NETLINK

The *type* indicates the communication semantics. Some defined types are:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

Last but not least, the *protocol* which defines a particular protocol used by the socket but usually only one exists for the specified *domain* and *type*¹⁰. Some common known combinations are:

- *TCP* = PF_INET with SOCK_STREAM
- *UDP* = PF_INET with SOCK_DGRAM

3.7.2 Named Pipes

Named-pipes, also known as *FIFOs*, are a very basic Unix example of an IPC mechanism. So for creating such a pipe we use the system call *mkfifo* which has two parameters:

- file
- mode

While the *file* declares the name of the pipe, the *mode* sets the access privileges of this pipe¹¹. A process which has write access to this file can push something onto this pipe now. Another process which has read access to the

¹⁰<http://man7.org/linux/man-pages/man2/socket.2.html>

¹¹<http://linux.die.net/man/3/mkfifo>

file can read from this pipe and get the message from the other process. A very simple command line example show how it works 3.3.

Listing 3.3: Named Pipe Command Line Example

```
1 > echo hello > mypipe &
2 [1] 1669
3 > cat mypipe
4 hello
5 [1]+  Done  echo hello > mypipe
6
7 ...
8
9 > cat mypipe &
10 [1] 2154
11 > echo hello > mypipe
12 hello
13 >
14 [1]+  Done  cat mypipe
15 >
```

As the FIFO suggests, they are working with *first in first out* principle. Those FIFOs are *half-duplex*, that means that they can only be read once by a process. It is possible to have multiple processes which write into the pipe but only one process will read from it at the same time. While it is possible to have multiple processes which are reading from the same pipe, one should know that only one of those readers gets the data. More about pipes can be found here [2].

3.7.3 Binder

The *Binder*¹² is a Android specific IPC mechanism. In short it provides another possibility to let two processes communicate to each other like activities, services and content providers. One example of it is the intent mechanism. Since the core part of DAMN is running at a much lower level on the system, we can not use those IPCs.

3.8 Boot Process

Another interesting part where our tool also needs to hook into is the boot process. If an Android device gets started it will run the bootloader at first. Usually, the bootloader is closed source and provided by the manufacturer of the device. It will initialize some low-level hardware which differs from device to device and also takes care of recovery and fastboot or download mode. After loading the *initrd* into the RAM it triggers the Android kernel which itself performs additional hardware initialization and start the system. The boot process differs in detail from Android version, manufacturer and device

¹²http://elinux.org/Android_Binder

but should overall be quiet the same process. After mounting the root file system, it will start the init process. As Linux user will know, this process starts every process on a system on startup. On Android it is mostly defined in the *init.rc* script which also could be exist of multiple parts. It will start different scripts and services (eg. *adbd*, *rild* and the Zygote daemon) [12].

3.9 Zygote

To start an application, Android use the *Zygote* process. As described on the boot process above, this daemon process is directly started by the init process. Every application will be a fork of this process. This fork will have its own address space and all needed libraries of the new application will be loaded as well [8].

3.10 Android Tools

Android provides developers with a *SDK*(Software Development Kit) and a *NDK*(Native Development Kit). Both come with useful tools and two of them we want do show in detail.

3.10.1 ADB

The Android SDK provides various little tools which can be used in multiple ways. Eclipse integrates most of them in their *IDE*(Integrated Development Environment) but some of them are also useful on the shell. The most used tool in our project is the *ADB*(Android Debugging Bridge). It give use the possibility to interact with the device on the command line. Here some useful commands:

- *adb devices*
- *adb wait-for-device*
- *adb pull*
- *adb push*
- *adb shell*
- *adb root*
- *adb remount*
- *adb install*

Let us shortly describe what this commands can be used for. The *adb devices* will print a list of all connected devices to the computer. In addition to this, the next command option *wait-for-device* is a nice solution to wait for a device till its ready to receive commands over *adb*. This is very useful in scripts where we have to wait until a device is ready to be used again after a performed command which causes the device to interrupt the *adb*

connection. *Pull* and *push* can be used to transfer files between device and computer. It is also possible to get a remote shell with the *shell* command option where we have access to the device as the shell user. On rooted devices it is also possible to get privileged access over adb with the *root* option. This will start also the remote shell with root access. Another useful option is *remount* which can remount Android partitions to gain write access to it. This is useful if we want to push data onto the */system* partition as it standardly mounted with read only access. The last command option we want to introduce is *install*. With this command we are able to install an apk file directly on the connected device.

There are other features as well that can be explored on the command line with the *help* option or on the developer page¹³.

ADB consists of two parts. The one we described until now is the part that is executed on the remote device, in our case the computer. The other part has to run on the Android device as a daemon process which directly gets started from the init process. This daemon is running with the name *adb* and starts very early in the boot process¹⁴.

3.10.2 NDK-Build

Most Android applications are written in Java but it is also possible to implement native code like C & C++. To achieve this, we need to download the NDK¹⁵ and setup Eclipse. As we wrote a short script that builds the native modules of our tool and pushes it immediately onto the device as well, we want to take a look at the *ndk-build* tool.

The *ndk-build* is a script provided by Android which is used to build native components of Android applications. To get *ndk-build* to work, it need some additional informations about our modules as well as some linker informations. This informations are stored in the *Android.mk* file¹⁶. In addition to this file we wrote a little shell script that build our native components and pushes it also on the device. This script will be explained here 8.1.

¹³<http://developer.android.com/tools/help/adb.html>

¹⁴https://events.linuxfoundation.org/images/stories/pdf/lf_abs12_kobayashi.pdf

¹⁵<http://developer.android.com/ndk/downloads/index.html>

¹⁶http://developer.android.com/ndk/guides/android_mk.html

Chapter 4

Reverse Engineering

Reverse engineering or reversing is the process of getting readable or manipulatable source code out of an executable application without having the source code available [3, Chapter 6]. The aim is it to get a deeper understanding about the application and how it performs. To reverse an application on Android we need the apk file. We can get this file from the device where it is installed or directly from the Play Store with this *python*¹ command line script². There are also other sources to get applications like for example from developer or project pages themselves. Anyway, once the apk file is available we can start reversing it with different tools.

We start to describe the reversing tools for Android applications. After a short investigation we analyze the different outcome of some of them and select one which we will use for DAMN. Afterwards we will take a deeper look into obfuscation that has the aim to make the reverse engineering process harder and the generated source code more difficult to understand. Finally we take a look at the limitations of those reverse processes.

4.1 General Reverse Engineering Term

Before we start to take a look at the different reversing techniques, we want to define the term of reverse engineering more detailed. Reversing and decompiling are often used in the same context and often understood to be the same. But this is not the case. Reverse engineering is a general term that describes the process of getting information about how something was built and to rebuild or improve it. This can be a simple thing as unpacking packages and take a look into it or, a more complex task, getting source code with specialized tools from the application which is called decompiling.

¹<https://www.python.org/>

²<https://github.com/egirault/googleplay-api>

4.2 Disassembler

A disassembler does the opposite part of what an assembler does. It reverses the executable file into assembler code. Although this code is readable, it is pretty hard to investigate because it consists of single instructions which are very related to the processor instruction set. On the other hand, disassemblers have a higher chance to alter code and reassemble it again to get a runnable application again. This is much harder if we take a look at the decompiler in the next section, because they are losing much more information at the reversing process.

The disassembled code on Android is called smali code. There are different tools which do disassembling and reassembling of executable files which we show below.

4.2.1 Baksmali

One Android disassembler is called *baksmali* and its counter part, an assembler is called smali. Those two tools also allow to disassemble and reassemble dex files. It has some similarities with the *Apktool* and is available on this github project³.

4.2.2 Apktool

Apktool is a tool to disassemble Android applications [11] into smali code. The generated assembler code can be changed and repacked to a running application. An additional feature makes it possible to debug the generated smali code⁴.

4.3 Decompiler

To get Java code out of an application we have to use a decompiler. In contrast to disassemblers, they are producing a higher level language which is more easily readable. There are some Java decompiler tools freely available on the Internet. They all reverse byte code into source code which can be reviewed or analyzed afterwards. Advanced decompilers are also able to reconstruct code sections which got optimized by the compiler and give it back in a well structured and formatted source code. In addition to such Java decompilers, there are also Android decompilers available which we list afterwards.

³<https://github.com/JesusFreke/smali>

⁴<http://ibotpeaches.github.io/Apktool/>

4.3.1 JD-Core

The first decompiler is called *JD-Core*. This decompiler brings also a graphical interface that allows to directly investigate the decompiled source code. It also supports multiple platforms where this decompiler can run on [11]. There are a few sub projects with this decompiler includes, but not limited, to a command line tool which can be used in scripts. Although it supports multiple platforms they are not supporting the ARM architecture. Because the source code is closed⁵ we are not able to compile it for those architecture on our own. This limits the usage of this decompiler.

4.3.2 JAD

JAD is another Java decompiler implementation⁶. It also provides a command line interface that makes it handy for scripts. Unfortunately this project is not maintained anymore and compared with *JD-Core* it produces less readable source code.

4.3.3 Android Decompiler

Unlike *Java*, Android uses *dex(dalvik executable)* files instead of *jar* files to run applications. The dex file contains all Java class files in a defined structure [7]. In addition to this file, there are also other resources stored into the *apk* file which containing additional information about the application like the manifest file [11].

Those specialized Android decompilers can be seen as additional converting tools. They can not reversing an application into source code but provide us with the possibility to use standard Java decompiler afterwards to do so. The exception proves the rule, *jadx* is the only Android specific decompiler which also produces source code. Here is a list we investigated:

1. Dex2jar
2. DAD
3. JEB
4. Dare
5. Jadx

Dex2jar

Dex2jar is a decompiler project that can convert Dalvik byte code into Java byte code. This makes it possible to use usual Java decompiler [11]. This step is necessary because Android uses a different runtime as Java 3.5.3. It

⁵<http://jd.benow.ca/>

⁶<http://varaneckas.com/jad/>

also supports disassembling & reassembling smali code. More information to this project can be found on this github repository⁷.

DAD

The Andrugard developer team implemented the *DAD* (DAD is A Decompiler) decompiler in python which is an open source tool. It is also capable for disassembling and reassembling of applications⁸.

Ded

The *ded* [9] project also has the aim to decompile Android applications and generate class files out of the dex file⁹. This project has been replaced with *Dare* that we will describe bellow.

Dare

As indicated above, *Dare* is the successor of *ded*. It is supposed to retarget applications from the apk or dex format into class files [17]. The project is open source and available here¹⁰.

JEB

JEB is a commercial decompiler which is available here¹¹. As it is commercial we did not investigate it in detail. Furthermore the source code of this decompiler is not available.

Jadx

A more comparable Android decompiler to Java decompiler is *jadx*. In differ to other Android decompiler this one produce directly source code without using a Java decompiler. It has a graphical user interface to investigate directly reversed code. The project can be found here¹².

4.4 Analyzing Reversed Source Code

We investigated the different decompilers to make a decision based on the reconstructed source code. We have chosen two decompilers which we will analyze further. The first one is the Java decompiler JD-Core. To get source code out of an apk file we have to use an additional tool to get the class

⁷<https://github.com/pxb1988/dex2jar>

⁸<https://github.com/androguard/androguard>

⁹<http://siis.cse.psu.edu/ded/index.html#banner>

¹⁰<http://siis.cse.psu.edu/dare/index.html>

¹¹<https://www.pnfsoftware.com/>

¹²<https://github.com/skylot/jadx>

files. This step is done with the dex2jar tool. The second decompiler is the Android decompiler jadx. As it can directly perform reversing on a apk file we do not need any additional tools. We use an Android application which was obfuscated and reversing the same class with both decompilers. The next two listing shows the output of them.

Let us start with dex2jar combined with JD-Core:

Listing 4.1: Reversed Source Code with Dex2jar and JD-Core

```
1 ...
2
3 private static final String ARG_GAME_ID = "ARG_GAME_ID";
4 private static final String ARG_PAUSED = "ARG_PAUSED";
5 private static final String ARG_QUESTION_INDEX = "ARG_QUESTION_INDEX";
6 private boolean mCountDownRunning = false;
7 private String mGameId;
8 private QuestionFragmentInteractionListener mListener;
9 private View.OnClickListener mOnClickListener = new View.OnClickListener
    ()
10 {
11     public void onClick(View paramAnonymousView)
12     {
13         long l1 = Calendar.getInstance().getTimeInMillis();
14         int k;
15         if (paramAnonymousView.getId() == QuestionFragment.this.mViews.btn0.
            getId())
16             k = 0;
17         long l2;
18         while (true)
19             {
20                 l2 = l1 - QuestionFragment.this.mTimeStampStart;
21                 QuestionFragment.access$402(QuestionFragment.this, false);
22                 Iterator localIterator = QuestionFragment.this.mViews.buttons.
                    iterator();
23                 while (localIterator.hasNext())
24                     ((AnswerButton)localIterator.next()).setOnClickListener(null);
25                 if (paramAnonymousView.getId() == QuestionFragment.this.mViews.
                    btn1.getId())
26                     {
27                         k = 1;
28                     }
29                 else if (paramAnonymousView.getId() == QuestionFragment.this.
                    mViews.btn2.getId())
30                     {
31                         k = 2;
32                     }
33                 else
34                     {
35                         int i = paramAnonymousView.getId();
36                         int j = QuestionFragment.this.mViews.btn3.getId();
37                         k = 0;
38                         if (i == j)
39                             k = 3;
```

```

40     }
41   }
42   QuestionFragment.this.answerQuestion(k, 12);
43 }
44 };
45
46 ...

```

The second snippet shows the same reversed code section with the jadx decompiler:

Listing 4.2: Reversed Source Code with Jadx

```

1 ...
2
3 private static final String ARG_GAME_ID = "ARG_GAME_ID";
4 private static final String ARG_PAUSED = "ARG_PAUSED";
5 private static final String ARG_QUESTION_INDEX = "ARG_QUESTION_INDEX";
6 private boolean mCountDownRunning = false;
7 private String mGameId;
8 private QuestionFragmentInteractionListener mListener;
9 private OnClickListener mOnClickListener = new OnClickListener() {
10     public void onClick(View view) {
11         long timeStampEnd = Calendar.getInstance().getTimeInMillis();
12         int chosenAnswer = 0;
13         if (view.getId() == QuestionFragment.this.mViews.btn0.getId()) {
14             chosenAnswer = 0;
15         } else if (view.getId() == QuestionFragment.this.mViews.btn1.getId())
16             {
17             chosenAnswer = 1;
18         } else if (view.getId() == QuestionFragment.this.mViews.btn2.getId())
19             {
20             chosenAnswer = 2;
21         } else if (view.getId() == QuestionFragment.this.mViews.btn3.getId())
22             {
23             chosenAnswer = 3;
24         }
25         long answerTimeInMs = timeStampEnd - QuestionFragment.this.
26             mTimeStampStart;
27         QuestionFragment.this.mCountDownRunning = false;
28         for (AnswerButton button : QuestionFragment.this.mViews.buttons) {
29             button.setOnClickListener(null);
30         }
31         QuestionFragment.this.answerQuestion(chosenAnswer, answerTimeInMs);
32     }
33 };
34
35 ...

```

We can see that the first listing has problems with the naming and the structure. Jadx reverses the source code in a much cleaner and more readable way because it has more information available which does not get lost during the use of dex2jar as JD-Core uses it. Because of this we are using the Android decompiler jadx in our script which we are providing 8.5.

But as it is one simple script it can be easily adopted if there would be even better tools for reversing.

4.5 Obfuscation

We have described what a decompiler is and what we can do with it. The potential to use this tools in a malicious way is manifold. For example, an attacker could use this gathered information to find security leaks and use those for its own purpose or use it to rebuild a clone of an application that either use features of the origin or tries to trick users to use them to gain information. A way to protect source code is to use *obfuscation*.

4.5.1 How it Works

Obfuscation is a way to rearrange the source code in a certain way that it is still executable but very hard to read and therefore difficult to investigate. There is also a technique to obfuscate on byte code level. If we talk about obfuscater in this thesis, we will mean the one which alters the source code. But there are also techniques to make it almost impossible to decompile. The following list will show possible obfuscation techniques and explain them afterwards:

- Layout Transformation
- Control Transformation
- Data Transformation
- Prevent Transformation

Layout Transformation

This step removes the source code formatting information that are on most class files. This can not be recovered by a decompiler. It also renames identifiers such as classes, methods, fields and variable names and removes comments in the source. But this has a rather low impact onto decompiler and is only the first transformation technique.

Control Transformation

This transformations are using a repertoire of three basic features. The first is *aggregation* which inlines methods, outline statements, clone methods and unroll loops. Second feature is the *ordering* that will reorder statements, loops and expressions. The last one is *computation*, it changes table interpretations as well as it extends loop conditions or alters flow graphs.

Data Transformation

This can also be split into three groups. The *storage & encoding*, split variables, change encodings, alter scalars into object and convert static data into procedural data. This prevents searching for static strings like keys because they will be produced every time they are needed. The next group of operations called *aggregation*, they merge scalars, modify relations and split, fold or merge arrays. The outcome of those techniques can be quite confusing. The last one is called *ordering* which performs reordering of variables, methods and arrays. All those operations have the purpose to make it hard to read the source code after decompiling.

Prevent Transformation

A completely different way of obfuscation is the *prevent* transformation. They do not obfuscate the code in a way that they are hard to read for humans but they try to prevent the decompile of the source code. This can be achieved by adding additional information that will let the decompiler crash but have no effect on the application itself. There are two possible ways, either exploring weaknesses of a certain decompiler tool or knowing problems with a general decompiling technique.

4.5.2 Summary Obfuscation Techniques

Those are the four basic obfuscation techniques which can be used by an obfuscator [4]. Not all of those transformations will be used on all obfuscation tools because some are more complex than others. We also want to give a list of a couple of obfuscation tools and a short description on how they are working in the next section.

4.6 Obfuscation Tools

4.6.1 ProGuard

Android provide an obfuscation tool named *ProGuard* which is included into the SDK. It is a free tool that is pretty easy to use and can perform those features:

- Shrink
- Optimize
- Obfuscate
- Preverify

Shrink

At this step ProGuard will analyze the source and remove unused code sections like dead code.

Optimize

Then it will do code optimizations which can change visibility of classes and methods. It can also remove unused parameters or inline methods into others which can lead to performance improvements.

Obfuscate

Next step is to rename classes, fields, methods as well as parameters.

Preverify

The last step verifies that all done changes make it impossible to break out of the sandbox.

ProGuard uses layout obfuscation which renames the packages, classes and variables. It also performs data obfuscation by converting static data into procedural data. That means that it will create a method which return the origin information generated by code. This makes it hard to detect strings with specific information as static passwords or URLs [16].

4.6.2 DashO

Another obfuscation tool is called *DashO* which is a commercial software from *PreEmptive*. It can perform layout, data and control flow obfuscation [16]. In the data obfuscation step, in contrast to ProGuard, it can perform string encryption. This is an advanced possibility to make it harder gaining information out of static data like access keys. It also tries to prevent decompiling to gain a higher benefit from this tool and preventing decompilers to work proper.

4.6.3 Decompiled Obfuscated Source Code

To get a more detailed imagination about how a reversed obfuscated source code could look like, we provide an example of a reversed application:

Listing 4.3: Reversed Obfuscated Source Code Snipped

```
1 package com.a.a.a;  
2  
3 import android.os.FileObserver;  
4 import android.os.Handler;  
5 import android.os.Looper;
```

```
6 import a.a.b.m.FileInfo;
7 import a.a.d.m.FileMetaData;
8 import a.a.f.p.StorageProvider;
9 import a.a.f.p.b;
10 import java.io.File;
11
12 public class g extends FileObserver
13 {
14     private static long c = 0L;
15     private Handler a;
16     private String b;
17
18     ...
19
20     public void onEvent(int paramInt, String paramString)
21     {
22         int i = paramInt & 0xFF;
23         switch (i)
24         {
25             default:
26             case 1024:
27             case 2048:
28             case 2:
29             case 256:
30                 long l1;
31                 long l2;
32                 do
33                 {
34                     return;
35                     a();
36                     return;
37                     l1 = System.currentTimeMillis();
38                     l2 = l1 - c;
39                 }
40                 while ((c != 0L) && (l2 <= 500L));
41                 c = l1;
42             case 4:
43             case 64:
44             case 128:
45             case 512:
46             }
47             a(paramString, i);
48         }
49
50     private static class a
51         implements Runnable
52     {
53         private String a;
54         private String b;
55         private int c;
56
57         a(String paramString1, String paramString2, int paramInt)
58         {
59             this.a = paramString1;
```



```

60     this.b = paramString2;
61     this.c = paramInt;
62 }
63
64 public void run()
65 {
66     String str = this.a + File.separator + this.b;
67     FileInfo localFileInfo = FileInfo.getFileInfoForProvider(1, str);
68     if ((2 == this.c) || (4 == this.c))
69     {
70         File localFile = new File(str);
71         if ((localFile.exists()) && (localFileInfo != null))
72             localFileInfo.setProviderMetaData(new FileMetaData(localFile.
lastModified(), localFile.length(), "", localFileInfo.mName));
73         return;
74     }
75     StorageProvider.G().f(null);
76 }
77 }
78 }
79
80 ...

```

This is a moderate obfuscated class where we can see that classes, methods and variables are renamed as well as that the general structure looks shredded. But at least it is readable source code again. There are also other decompiled parts that can not be recovered like this:

Listing 4.4: Failed Reversed Obfuscated Source Code Snipped

```

1 ...
2
3 // ERROR //
4 public static void a(Context paramContext)
5 {
6     // Byte code:
7     // 0: ldc 2
8     // 2: monitorenter
9     // 3: getstatic 56 a/a/b/f:e Landroid/content/Context;
10    // 6: ifnonnull +11 -> 17
11    // 9: aload_0
12    // 10: ifnull +46 -> 56
13    // 13: aload_0
14    // 14: putstatic 56 a/a/b/f:e Landroid/content/Context;
15    // 17: getstatic 36 a/a/b/f:b Landroid/security/IKeystoreService;
16    // 20: instanceof 48
17    // 23: ifeq +33 -> 56
18    // 26: new 217 java/io/File
19    // 29: dup
20    // 30: getstatic 56 a/a/b/f:e Landroid/content/Context;
21    // 33: ldc 219
22    // 35: iconst_0
23    // 36: invokevirtual 225 android/content/Context:getDir (Ljava/
lang/String;I)Ljava/io/File;

```

```

24 // 39: ldc 227
25 // 41: invokespecial 230 java/io/File:<init> (Ljava/io/File;Ljava
/lang/String;)V
26 // 44: astore_2
27 // 45: invokestatic 232 a/a/b/f:b ()I
28 // 48: ifne +12 -> 60
29 // 51: aload_2
30 // 52: invokevirtual 236 java/io/File:delete ()Z
31 // 55: pop
32 // 56: ldc 2
33 // 58: monitorexit
34 // 59: return
35 // 60: getstatic 38 a/a/b/f:c La/a/b/JBKey;
36 // 63: iconst_1
37 // 64: getstatic 40 a/a/b/f:d Ljavax/crypto/spec/IvParameterSpec;
38 // 67: invokevirtual 241 javax/crypto/spec/IvParameterSpec:getIV
() [B
39 // 70: invokevirtual 247 a/a/b/JBKey:initCipher (I [B)V
40 // 73: new 158 java/io/OutputStream
41 // 76: dup
42 // 77: new 249 javax/crypto/CipherOutputStream
43 // 80: dup
44 // 81: new 251 java/io/FileOutputStream
45 // 84: dup
46
47 ...
48
49 //
50 // Exception table:
51 // from to target type
52 // 73 116 119 java/lang/Exception
53 // 3 9 135 finally
54 // 13 17 135 finally
55 // 17 56 135 finally
56 // 60 73 135 finally
57 // 73 116 135 finally
58 // 121 132 135 finally
59 // 142 151 135 finally
60 // 60 73 141 a/a/b/JBException
61 }
62
63 ...
64

```

The decompiler was not able to decompile this method. But it shows the instructions which can also be used to get information about what this method is doing. However, it takes quite longer to read and understand this instead of the higher level source code above.

4.7 Used Decompiler in DAMN

As reading through source code gives always a benefit whether it is obfuscated or not, we also use a decompiler to reverse the source code out of an application and show it in the tool. Unluckily, most of the decompilers are closed source or using libraries which are not compiled to work on ARM architectures. Given to the fact that most Android devices use this processor architecture, we have to use another way of providing this functionality.

For usability reasons we decided very early that we will use a browser interface to interact with the device. Luckily most devices like notebooks are not using the ARM architecture and we can use a decompiler on those machines. To make it easy to perform this manual step on the other machine, we wrote a script which runs on Linux and reverses the source code of a specific application and move the data in the correct directory where DAMN can access it 8.1.

4.8 Limitations

As described above, if an obfuscation technique is used which makes it impossible for the decompiler to reverse the object code back into source code, so we can not make use of it. On most applications we reversed it is only a subset of methods where the decompiler does not work properly.

Another bigger limitation is the use of native compiled libraries that are used by the application we want to investigate. Currently we do not provide the possibility to decompile native libraries as well as we do not offer a way of dynamic analyzing those files. The only way we provide to investigate this libraries is to take a look into the interface by hooking the JNI interface methods.

4.9 Analysis Methods

In addition to this chapter we shortly want to give another notation. As reversing provides a way to gain the source code of any closed application, one can use this information to get furthermore information about it. This can be done with static analysis.

4.9.1 Static Analysis

Static analysis, also known as *static code analysis*, is the process of analyzing source code of a program which is not executed. This analysis technique gives an overview about the structure of the project and can help to determine where from the security perspective the critical parts are. What it makes pretty hard to analyze is obfuscated code because it adds more complexity into it [5].

4.9.2 Dynamic Analysis

Another technique is called *dynamic analysis*. As the name suggests, this will perform its analyzing methods on a running application. In differ to static analysis it does not have any issues with obfuscated source code since the application has to be executable regardless whether it is obfuscated or not. The downside is, that this analyze technique relies on the input an application receives during the analysis. If not all possible opportunities were executed, it will not find untouched methods and therefore can not analyze them. Furthermore, dynamic analysis can only run on prepared environment or application which allows it to obtain what the investigated application does. Dynamic and static analysis are often combined to be more effective [5].

Chapter 5

Concept

5.1 State of the Art

There are two basic ways of analyzing an application. Either static or dynamic. While some tools combine them, they are most likely fully automated and do not a way of providing manual investigation features. Sometimes manual investigation is required to find some conceptional security flaws that can be abused by others.

Analyzing an application only statically does have some drawbacks. One of them is that it is hard to detect dynamic behavior of malware because it is not executed. There is also the threat that reverse engineered source code is potentially incomplete and can not be analyzed proper. Another potential problem is the obfuscation of source code. It makes it hard to perform investigations of the reversed source code for humans because it is hard to read.

Dynamic analysis on the other hand does not struggle with obfuscation. As the application still has to be executable, it does not have a big impact. Another great opportunity is to test the behavior of a running application with different parameters. This manipulations can give new ways of investigating security flaws or stability problems.

5.2 Goals

The aim of DAMN is to combine those two features of analyzing applications to provide a superior solution for manual investigations. To accomplish this, we have to overcome the biggest issue of nowadays applications from the reverse engineering point of view, obfuscation. Because obfuscation can perform structural changes to the source code, it is way more effort needed to investigate the source code. Dynamic analysis can deal with this and our tool takes use of that.

If someone wants to take a deeper look into an application, it makes sense

to explore the features it provides by using it. With security in mind, one can find security concerns on potentially insecure interactions such as a login. Although we can find it easily by navigating through the application, we do not have any hints on the source code. Searching for strings is pointless if a good obfuscater is used. DAMN brings those two parties together and makes it able to find those interesting source code parts during the application is executed. This makes it easy to find the source code which is related to an interaction on the running application and save time although the source code is obfuscated.

Another benefit of combining the dynamic analysis method with the static source code for manual investigations is the opportunity to test the behavior of an application under changed circumstances during the execution. Instead of investigating various classes and methods which were parted by the obfuscater, we can test the flow of an application with different parameters and how it is reacting on it. This helps to save time because we can directly test if the application act as it should or have a potential flaws.

5.3 Scope of DAMN

Most of the analysis tools that combine both, static and dynamic analysis features, are half or full automated analyzing tools. DAMN distinguish from this approach. Since it is sometimes needed to investigate an application manually, we want to give support for this purpose. Especially when it comes to obfuscated source which does not add further security to the application but has the aim to protect the source code. Because of this manual investigations are more complex. Although reversing is still possible with obfuscated source it makes it pretty hard to find security flaws manually. DAMN helps to overcome this complexity and makes obfuscation almost neglectable for manual investigations.

Furthermore it gives the opportunity to manipulate the running application without alter the source code. This can be used to determine the behavior of an application with different values. It is also planned to use behavior rules which are a way to automate some manual manipulation operations.

However, DAMN does not provide any automated analyzing features as it is developed to support manual investigations.

5.4 Use Cases

We will describe three use cases in which we will picture how DAMN can be used to investigate applications.

5.4.1 Security Researcher

Our first use case is the security researcher who wants to analyze applications to find potential security flaws and report them. Although this is done mostly with automated test suites to investigate as many applications as possible, manual investigations can help to gain further information. DAMN can help to reduce the time needed for those manual investigations as it is possible to seek through the running application as usual until risky parts have been found as for example server communications. The researcher can pause the application at this point and move stepwise through the methods. The dynamic part of DAMN will display which methods are used and what parameters are passed. In addition to that it is possible to manipulate the application to make further analysis.

Furthermore DAMN can be used on obfuscated applications to exhaust the full potential of it and helps to shorten the investigation time on those applications.

5.4.2 Software Development Company

Bigger software companies that do commercial software development usually have a separate test department to ensure a high quality of their products. Even though the company has the source code available without reversing they perform black box reversing as well. With DAMN they have the possibility to investigate their application in a proper time and get an idea of which information can be extracted from others. This helps to tighten security guidelines and prevent attacks from which everyone profits.

5.4.3 Malicious Attacks

Of course we also illustrate the other side of the opportunities DAMN provides. A potential attacker can make use of DAMN to investigate third party applications with vicious aims. Such investigations with malicious intentions are not new. The possibility to gain information about potential targets with reversing approaches is commonly used to find security flaws in applications which than can be used to do harm.

5.5 Summary

The use of reverse engineering tools such as DAMN is not reserved to such attackers anymore and can be used to get insights into the information flows of Android applications. This information can be used to improve security of applications and makes it harder for attackers to attack them.

Chapter 6

Tooling

6.1 SuperSU

Android provides different security mechanism which has the purpose to prevent abuse. On the other hand, it is also open to get extended by applications which offer new features. Some of those features need some additional permissions that the normal API is not offering. To get those applications still working, we can make use of the *root* privilege. Root give a process or a user the possibility to do almost anything on the system. Some of the system applications are using root to perform tasks which would not work without this privilege like Zygote.

Typically this privilege is only accessible for system components and can not be used by applications on Android. But there existing solutions for that problem. One of them is called *SuperSU* and is provided by *chainfire* ¹. Because this is a basic feature DAMN needs to work proper, we will have a deeper look into it.

6.1.1 Installation

There are multiple ways to install SuperSU onto the device. If the device is already rooted, the simplest one is to download it from the *Play Store*, install it and after a reboot everything should be done. Another solution is to download an update zip which can be flashed by the recovery. Such a file can be found here².

6.1.2 Installation Process in Detail

If we take a closer look at the update zip file and open it, we can find a few files and scripts which perform the installation in the recovery. As in every update file, there is a *update-binary* and a *updater-script*. In this case

¹<https://play.google.com/store/apps/details?id=eu.chainfire.supersu>

²<https://download.chainfire.eu/696/SuperSU/UPDATE-SuperSU-v2.46.zip>

the *update-binary* is a shell script which performs the installation. Luckily *chainfire* put also a lot of comments into this script which makes it easier to analyze the single steps.

The interesting parts are where SuperSU copies the files which are needed to work this mechanism and how it achieves the autostart from boot. The parts that are installed differs from the architecture and the Android version. Let us assume we look at a ARM device with 4.4.2 and SELinux enforced like our test device.

Here are the files which gets installed on the file system:

File	Directory
99SuperSUDaemon	/system/etc/init.d/
install-recovery.sh	/system/etc/
install-recovery.sh	/system/bin/
su	/system/xbin/su
su	/system/xbin/daemonsu
su	/system/bin/.ext/.su
supolicy	/system/xbin/
libsupol.so	/system/lib/
chattr	/system/xbin/
app_process	/system/bin/
Superuser.apk	/system/app/

The basic SuperSU binary files are *su*, *daemonsu* and *.su* which are basically all the same binary file. As it starts as a daemon process, which is required if SELinux is enforced, we have the following three options: *install-recovery*, *app_process* or over the *init* process *99SuperSUDaemon*. The *Superuser.apk* will be copied as a system application. *Supolicy* and *libsupol.so* are used to set the right policies for SELinux. The *chattr*³ binary is used to remove the immutable flag from a EXT2 file.

6.1.3 Usage

If the installation was successful, every time a process is requiring root privilege the SuperSU application will be notify the user and he can choose if it will be allowed or not. There are also settings to change this behavior and to define rules on further access.

6.2 Dynamic Manipulation

Another big part in our tool makes use of dynamic manipulation of applications. The idea behind such a feature is to hook into specific parts of an application and change values or the behavior of it. In general this tools

³<http://linux.die.net/man/1/chattr>

are built to make changes to the system or application for either adopt new features, alter the behavior or manipulate the look and feel. Although those tools are not built to use it for dynamic analysis, they can be used for it. On Android there are two tools which can be used for that. Both tools are made individually and are not related but have much in common.

6.2.1 Cydia Substrate

was developed by *saurik* also known as *Jay Freeman*. The whole project is closed source which makes it hard to make changes but there is a good documentation online available⁴. In differ to *Xposed* 6.3, Cydia Substrate was developed for both, Android and *iOS* and has the opportunity to handle native libraries as well. This is not restricted to the native parts of applications, it also can be used to investigate any native system part.

Unfortunately this project does not get as much attention as it should and saurik did not released updates for Android with newer versions as 4.3. This was the main reason why we decide to not use this framework for our tool.

6.2.2 Xposed Framework

The second tool which allows to manipulate applications dynamically is called the *Xposed* framework. It is developed by *rovo89* and *Tungstweny*. In differ to Cydia, Xposed is open source and can be investigated here⁵. This tool is only available on Android but is maintained actively. It now supports the latest Android version with ART and this is also the reason why we decided to use this framework for DAMN. A downside of the framework is, that it cannot handle native compiled application parts. It can only hook into the JNI interface where the native libraries gets called and change the parameter or the returned values of it.

Installation

The Xposed running, we require a rooted device and the installation of the *Xposed Installer*⁶. With this application it is possible to install the framework onto the device and handle the modules that are installed.

How Does Xposed Work?

We described the Zygote daemon previously which start every application with a fork of itself 3.9. Xposed will extend the libraries which gets loaded on start with an additional one called *XposedBridge.jar*. Because this library

⁴<http://www.cydiasubstrate.com/>

⁵<https://github.com/rovo89>

⁶<http://repo.xposed.info/module/de.robv.android.xposed.installer>

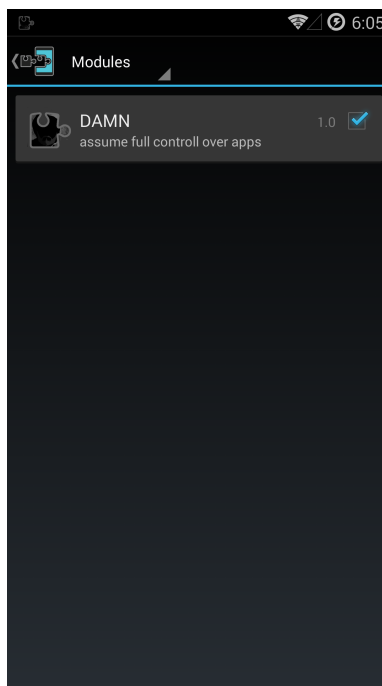
run in the same context as the new application, it is possible to interact with it and hook in those methods via Reflections 3.3.1.

Modules

Modules are independent applications which have the possibility to hook into an application and alter them. They will be shown up in the Xposed Installer as soon as the application with the module is installed. A detailed documentation about that can be found here⁷.

DAMN uses such a module that gets loaded on application start too. An interesting fact we should have in mind is that we can use multiple modules for the same application at the same time. This means if we want to use DAMN we do not have to be the only module for this application. On the other hand it may make not much sense to alter an application with another module and investigate its behavior. Turning other modules off is therefore the recommended way. This should be no problem because Xposed provides a user interface where single modules can be activated or deactivated 6.1.

Figure 6.1: Xposed Module Interface



⁷<https://github.com/rovo89/XposedBridge/wiki/Development-tutorial>

Usage

The normal usage of Xposed is to hook into applications and change their behavior or look. This could be an emoji replacer for *Whatsapp* or an extended *UI* tweaker for *AOSP*(Android Open Source Project⁸) devices. A list of some modules can be found here⁹. Further modules are available on various sites.

Security

One short remark about security in Xposed. A module can control pretty much everything on a single application. That it only will change the look or add a certain feature to it does not mean that this will be the only thing it does or will do in the future. As we can see in our module we wrote for DAMN, we are not restricted to one application at all. Be sure to use only modules which can be trusted in.

6.3 Xposed

The Xposed framework is an interesting open source project which can change functionality of an Android application without changing anything on the apk itself. Furthermore it is possible to write multiple modules which are able to access and change the same application simultaneously. If we would change the source code we would not be able to combine different implementations without reediting it. A module can be easily switched on/off followed by a reboot. All changes which are done to the application happen on memory level. The framework basically loads an additional library when the Zygote 3.9 process gets started. This process starts every application with a fork and because the library is now on the class path of this process, we are able to interact in its context. They also provide a few useful functions for searching methods in classes. Xposed hooks them and makes it possible to read and manipulate parameters or return values. The framework does not provide the possibility to single step each line because it does not use any debugging symbols (which are located in the symbol table) which would be needed for this. The capability to go through the source line by line is impossible, but method wise is feasible.

Usually Xposed will be used to change some parts in an application to change its behavior. To make this possible, we have to hook into a method. This is typically done by looking into the reversed source code, selecting a method in which we are interested into and hook it with a module. The framework provides some helper functions to achieve this but behind those

⁸<https://source.android.com/>

⁹<http://repo.xposed.info/module-overview>

helper functions it uses Java reflections 3.3.1. We could also use reflections to find methods directly.

6.3.1 XC_MethodHook Class

If we have hooked a method, we will have two possibilities to interact with it:

- Hooking on Call
- Hooking on Return

Hooking on Call

The *beforeHookedMethod* gives us the opportunity to modify static fields or parameters that are passed before the method can handle them. This is the first entry point where Xposed can manipulate values. If the called method is returning, the second Xposed method will be called explained below.

Hooking on Return

The second entry point is the *afterHookedMethod* method. If a hooked method is returning, this method gets called. We have again the opportunity to change the values of fields or the return value at this point.

6.4 Jadx

Since our tool consists of both, a static and a dynamic analyzing part, we use jadx for our static part. It is a tool for decompiling Android apk files into source code. The functionality of jadx makes it possible to list all methods of the classes and get the source code out of them. We can freely search across this extracted source like we could do as a developer. The reversed source code is very readable compared with Java decompiler which we already investigated here 4.4.

This component run as a simple *BASH* script 8.5 on Linux-based shells. This is the only component in this project that do not run on the device(beside the JavaScript parts on the browser pages).

6.5 Civetweb

As we are using a second screen for investigating applications, we need a way of display those additional content. Because it should be highly available on nearly every device, we were looking for a browser page that fits for this purpose. To supply this, we are using *Civetweb*¹⁰, a lightweight C/C++ based

¹⁰<https://github.com/civetweb/civetweb>

web server that can be compiled for Android. It also supports SSL/TSL connections which are useful to protect the connection between device and browser. Moreover websockets are also supported. DAMN use them for sending data with a low latency that makes realtime monitoring and manipulation of applications possible. Secure websockets are capable of SSL/TSL encryption [23], which are forced to be used if we want to use HTTPS, since for security reasons, no downgrade of the protocol is feasible.

6.5.1 WSS

For the communication between the device and the page on the browser, we are using websockets. Because we are using HTTPS for our page, we also have to use the secure websocket as the browsers do not allow to use a unprotected communication ways on a protected page¹¹. In addition it provides a fast way of communication between our two systems [24] which is important to have as less overhead as possible.

¹¹<http://stackoverflow.com/a/9752145>

Chapter 7

DAMN

DAMN is our debugging tool for reversing obfuscated source code. It was built with obfuscation in mind and therefore it is a specialized reversing tool. It consists of multiple parts of open source projects which was another reason to give it back to the open source community as it would not exist without them. The different projects will be described here in greater detail. After describing the different parts of the tool itself to get a deeper comprehension how it works.

7.1 Overview

So far, we described the architecture of Android as well as some IPC mechanisms and the reverse engineering process. With the knowledge of the different components we described above, we now come to our tool itself. In the next sections we will explain detailed how it is exactly working and how the components are interacting with each other. First we start with the general structure, the architecture of it and after that we step gradually through all features of it. Additionally we do a test run with a simple application where we demonstrate that everything is working. After that, we also investigate two other applications and describe shortly what we have found with DAMN and how long it did take.

7.2 Architecture

Let us start with the architecture of the tool to get an overview about the interaction of the components and a general picture of it. To interact with the application on the device and also have the opportunity to control its behavior, we use a HTTP server on the device. This makes it possible to use the application normally and monitor it on a browser like we see on this figure 7.1. The advantage of using a browser is that it works on almost computers or tablets without any additional applications installed. DAMN

provides also two ways of connectivity between device and browser.

- USB
- WiFi

USB

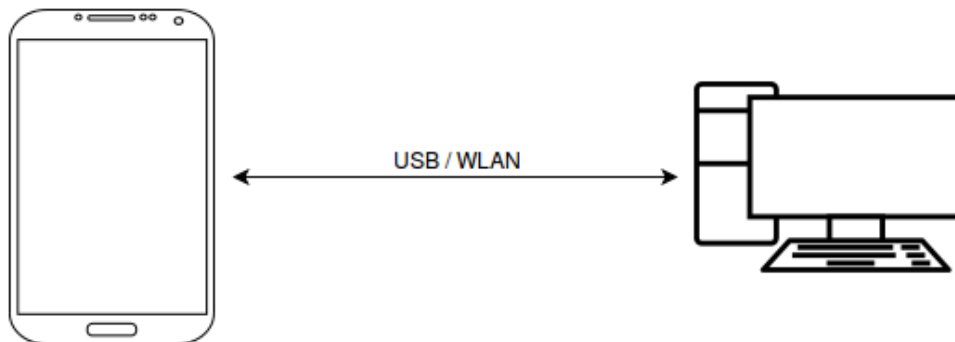
A shared library makes it possible to connect the device to another system via USB port. If someone wants to investigate an application which starts with the system itself (e.g. *NFC Manager*) we do not have a valid WiFi connection. The connection over the USB cable makes it possible to connect through the browser and control it right before a wifi connection starts.

Currently it is only integrated as a shared library but is not used in DAMN actively. To use this feature it has to be triggered or implemented in source code manually.

WiFi

The usual way of using DAMN is over a wifi network. If the device is connected to a wifi hotspot any system in the same wifi can connect to the device and therefore makes it possible to investigate applications through it. For more details on the protocol go to section 7.12.

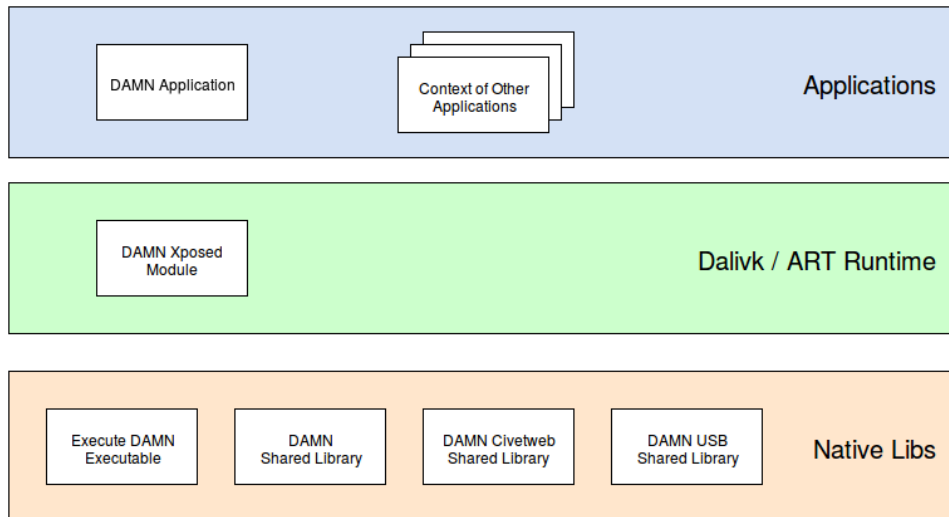
Figure 7.1: DAMN Components Architecture Overview



In the figure 7.2 we see it from the Android architecture perspective. We choose to splice the components into three layers which are referencing into the levels of the Android architecture:

- Application Layer
- Runtime Layer
- Native Library Layer

This layers point out how deep DAMN is interacting with the Android system as it is not only on the top layer as an application. Let us take a

Figure 7.2: DAMN Components Architecture Overview

closer look at the three layers.

7.2.1 Application Layer

In the application layer we find the DAMN application which provides a very basic user interface where user can select applications that they want to investigate through our tool. This part will be described more detailed here 7.4.1.

7.2.2 Runtime Layer

As the runtime differs from the Android versions it could either be Dalvik or ART which is running on the device. Thanks to Xposed it does not matter which runtime is used. The Xposed module will get loaded on the boot process along with Zygote. Once this module is loaded DAMN have the possibility to interact with every application on the device. More about the module can be found here 7.8.

7.2.3 Native Library Layer

The layer which is situated on a very low level on Android 3.5 us the native library layer. Components which are placed here can be started before any application itself gets started. This makes it possible to start the Civetweb component very early on the system with the advantage that we also can investigate applications which are started during the boot process (some

system applications). Combined with the USB library, we can connect a computer at this very basic stage where the device itself does not have any Internet connectivity. More information on this layer is described here 7.7.

7.3 Interaction Structure

Another point of view to DAMN is the interaction between the different components. The tool uses different features to interact through the different parts as it can be seen in figure 7.3. There are three basic interaction mechanisms we are using:

- File
- IPC
- WSS

7.3.1 File

A very simple opportunity is to share information across a file. While basically everything is a file on a *Linux* system, what we mean here is to write information into a basic file which can be read from other processes. This option is used to store configuration data which has to be accessible through the investigated application. Because we do not know which permissions the investigated application have, we need to provide the information in a way any application can access it.

As any application has access to its own directory where it is installed on (typically `/data/data/<package name>/`) we put this configuration file into this location. Because of the Unix file permission system, our DAMN application needs to create such a file with privileged rights. This is achieved through SuperSU 6.1 and the change of the accessibility mode of this file to all (`chmod 0666`). Anytime an application gets loaded, the DAMN Xposed module will check if a file with the name `damn_packages.pkg` exists on the root directory of the application. If this is the case, the file will get read and all relevant packages will be hooked. This process will be described in more details in this section 7.5.

7.3.2 IPC

While the transportation of information through a file, as described previously, can also be an IPC, we mean here the interaction with other processes on runtime. The conclusion is therefore that the communication through the file 7.3.1 is a static communication and the IPC we now describe is a more dynamic one.

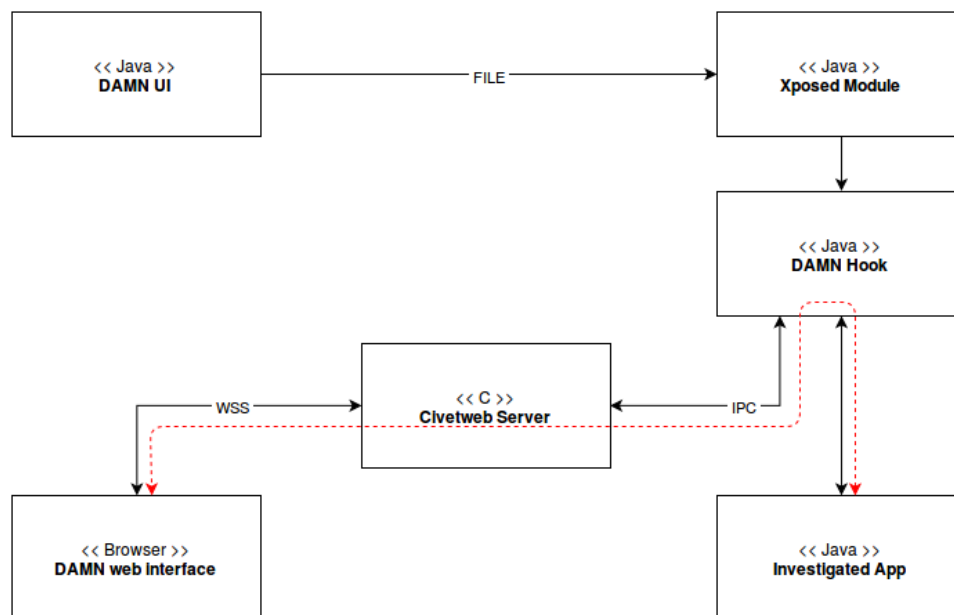
The IPCs we use here are the basic ones from the Android system 3.7.

7.3.3 WSS

As we decided to take a second screen solution to interact with the device to have a more comfortable possibility to control the investigated application, we also need a way to communicate to it. DAMN starts a simple HTTP server on the device and offers the opportunity to open a browser on a different system that is connected to the same network. The server is realized through Civetweb 6.5 as well as the secured web sockets (WSS 6.5.1).

We use the web socket to interact with the browser since it provides a fast way of communication. This is essential because it prevents overhead in the communication which are leading to lags in the application we investigating actively. Another advantage is that it is well supported in JavaScript. More about the communication between the device and the browser can be found here 7.12.

Figure 7.3: Interaction of Components Overview



7.3.4 Communication Trace

There is also a dotted red line in the figure 7.3 above. As previously mentioned, there is an active communication and a passive one in which only configured data is stored. The active communication happens on runtime and this is the way the communication will flow during the investigation of an application.

7.4 DAMN User Interface

Since DAMN has two parts where a user can interact with it, we split this section into the DAMN application user interface and the browser user interface. Both are needed to either set the settings to make an investigation or to monitor respectively manipulate, the behavior of the investigated application.

7.4.1 DAMN Application Activities

To start off with the user interface on the DAMN project, we first have a look onto the application. Once it is installed and configured correctly, we can take a look at the installed applications on the device through this application. If we found an interesting one, we can select it and get more information about it in a second activity. Let us take a closer look into the two activities.

List Activity

The first activity that will be opened on start is the list activity which is printed here 7.4 and here 7.5. It is a pretty simple *ViewPager* where the user has the choice between five different pages:

- Downloaded
- System
- Running
- All
- Track

The *downloaded* page will list all user installed applications, while the *system* page shows the applications. To differ between those two possibilities, we read the application info to get the *FLAG_SYSTEM*¹ value which indicates whether it is a system application or not. On the *running* page we have listed all running application on the device. As the Xposed module has to be loaded with the application we want to investigate, we have to be sure that the application does not run or gets restarted. This either can be made possible by restarting the device or by forcing to kill the application manually. Obviously, the *All* page shows all installed applications on the device. On the *Track* page we list all applications which the user decided to investigate.

Every application which is listed as an item can be clicked to get more details. This opens the detail activity which is described on the next subsection.

¹http://developer.android.com/reference/android/content/pm/ApplicationInfo.html#FLAG_SYSTEM

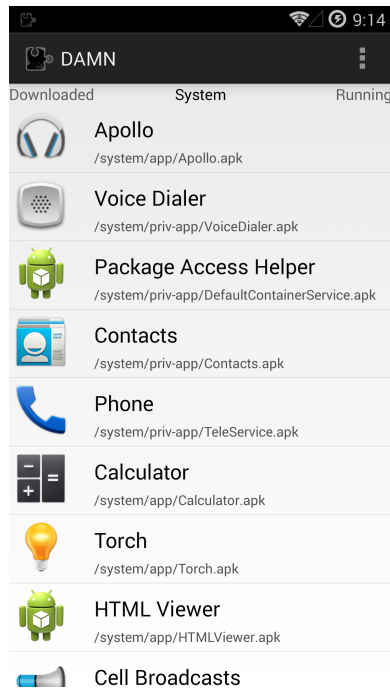


Figure 7.4: System Page

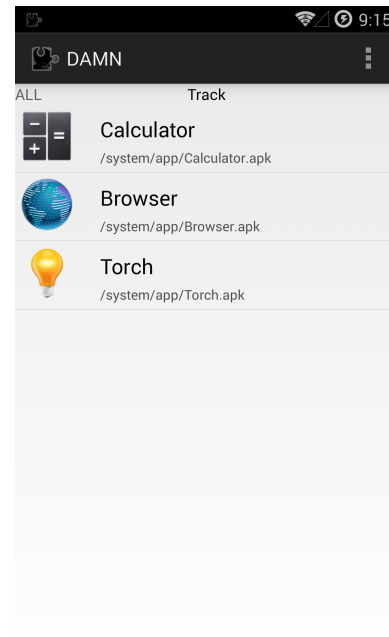


Figure 7.5: Track Page

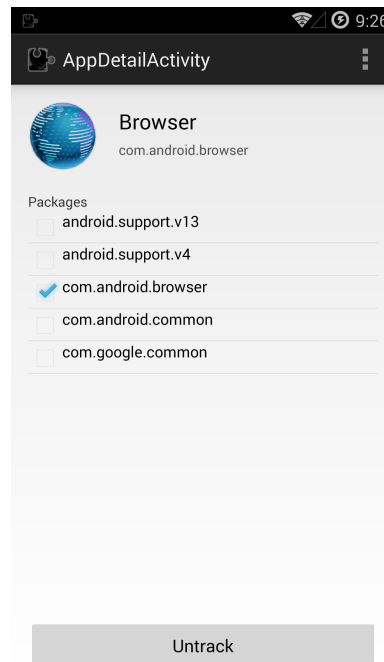
Detail Activity

On the detail activity we can take a look at the application we want to investigate. Besides the icon, name and the package, we also can see all internal packages of the application 7.6. All packages of the application will be divided into multiple packages if the first three sub-packages (separated by a dot) are different. Those divided package items can be selected individually and give a great opportunity to get more selective. This makes sense on applications where many packages are used from third party libraries which we do not want to investigate. Or we already have a clue where interesting things happen and do not want to have all other packages included too.

Here we can see five packages from this application but we only want to investigate one of these. With the button on the end we can either save our selection or *untrack* the application. The information will be stored on the package of the investigated application. More details about the configuration file for each application can be found here 7.5.

7.4.2 DAMN Browser Pages

Having a browser interface for interacting on an application we want to investigate is very useful. While we can handle the application on the device, we can get additional information on the browser. To use a browser also gives

Figure 7.6: Detail Activity

us the freedom of using this solution on almost any devices where a browser is installed on (and supporting web sockets and JavaScript). The next few subsections will show how we implemented it and how it can be used.

Get Connected

Before we take a look at some applications, we have to connect the device with the browser. The easiest way of archiving this is to be in the same wireless network on both, the device and the computer where we use the browser. If DAMN is running on the device, we only need to get the IP address of it. It can be found in *Settings* -> *About phone* -> *Status* -> *IP address*. On the browser we can now enter this address and we will connect to the device 7.4.3. After that we should be able to see the start page as in the next section will be described.

7.4.3 Start Page

The start page should be the first we can see on the browser to be sure that the connection works. On this figure 7.7 we can see how it looks like. Because of on the device we have a functional Civetweb server running, we will get a response on this address and will receive the start page. It does not matter if we call it directly with SSL/TLS or not since Civetweb redirects

it automatically to a secured connection.

Figure 7.7: Browser Start Page



While the page is processed by the browser, it also starts a web socket connection to the device to receive commands from it. The state of the connection can be directly seen on the start page which prints them directly on the page.

- Connect ...
- Connection error!
- Can't connect to device!
- Waiting for Applications

Connect

This will be printed when the web socket tries to connect to the device.

Connection Error

Only appears if something went wrong, either on the connection phase or on a later point of time.

Can't Connect to Device

If the browser can not get a web socket connection to the device, it will print this text. This should actually not happen since the displayed page also has the same origin except this ports are locked over the network.

Waiting for Applications

In case everything works and the connection is established, this text will be printed. This indicates that we can open now one of the application we want to investigate and let the browser open a new page which is shown on the next subsection.

No Web Socket Support

Of course we have to use a browser which supports web sockets. Luckily it is supported by many browsers nowadays². Even Android browsers support them on the latest versions. If a browser is used that do not provide web socket support, an alert will be raised which informs the user about it. Switching to a browser with proper web socket support is recommended.

7.4.4 Tracking Page

Now we can directly start to open an application which we want to investigate. This will trigger Civetweb to send a command to the start page with the purpose to open a new tab in the browser. This new tab is called *tracking* page and it is printed in this figure 7.8. In this example we use the *Torch* application from *CyanogenMod* version 11.

Figure 7.8: Browser Track Page

The screenshot shows a browser window with the title 'Tracked Application Control'. The page content is as follows:

```

private boolean torchServiceRunning(Context paramContext)
{
    Iterator localIterator = ((ActivityManager)paramContext.getSystemService("activity")).getRunningServices(100).iterator();
    while (localIterator.hasNext())
    {
        ComponentName localComponentName = nextservice;
        if ((localComponentName.getClassName().endsWith(".TorchService")) || (localComponentName.getClassName().endsWith(".RootTorchService"))) {
            return true;
        }
    }
    return false;
}

public void onReceive(Context paramContext, Intent paramIntent)
{
    {
        "class": "net.cactii.flash2.TorchSwitch",
        "method": "net.cactii.flash2.TorchSwitch",
        "parameters": [
            {}
        ]
    }
}

{
    "result": {},
    "method": "onClick"
}

{
    "fields": [
        {}
    ]
}

```

This page is more complex and has different sections for different information. Let us start with the top section. The first button in the *Con-*

²<http://caniuse.com/#feat=websockets>

trol section is the *play* button. The name already indicates that this will let the application run like as it would do without getting investigated by DAMN. The second button is the *step* button that makes it possible to iterate through the method calls as described here 7.9. On the top right corner, there is the *rule* button located. Here we can declare behavior rules which we will describe here 7.11.

The next section is the *Source Code* section. It shows the decompiled source code of the investigated application. Because the decompilation can not be automated on the device itself, it is possible that there is no source code available. In this case *no code available* will be displayed.

On the left side below there is the *Parameter* and the *Return Values* section. Both sections can be edited by the user and show which parameters or return value are passed through the method.

The only section left is called *Fields*. It shows all fields that are declared in the actual class. Although it is not implemented to alter those fields with DAMN, it would be no problem because the principle would be the same as on the two described fields previously.

7.5 Configuration File

The Android permission system restrict the file system access of an application to another. It is using the DAC mechanism for that 3.6.1. Because we can not be sure an application has the access permissions to read from the SD card, we have to find another way to store the individual configuration file for each investigated application. A very simple solution is to store the file on the base of the install directory of the application. The installation directory can differ but is mostly located on `/data/data/<package name>/`3.5.5. Because DAMN also underlies those regulations, we have to use SuperSU to write into this directory and change the access mode of the file. Here we have the steps from the code sniped that makes this possible:

Listing 7.1: Snipped of Saving the Configuration File

```

1 Runtime.getRuntime().exec(new String[]{"su", String.valueOf(
    applicationInfo.uid), "cp", tmp.getAbsolutePath(), target.
    getAbsolutePath()}).waitFor();
2 Runtime.getRuntime().exec(new String[]{"su", String.valueOf(
    applicationInfo.uid), "chmod", "0666", target.getAbsolutePath()}).
    waitFor();

```

The first command copies the temporary file with the configuration to the root directory of the investigated application. The second one changes the access mode to `0666` which let any application read and write this file. With those two commands, the investigated application can read this file which is important for the Xposed module that gets started when the application is loaded and run in the same context. This is a very necessary

part. As soon as the application is started, we are restricted to the same permissions that the application has. So we have to be carefully which resources we can access. The best concept is to guess that the application has no additional permissions we can use and therefore we only can use the very basic permission each of the applications have.

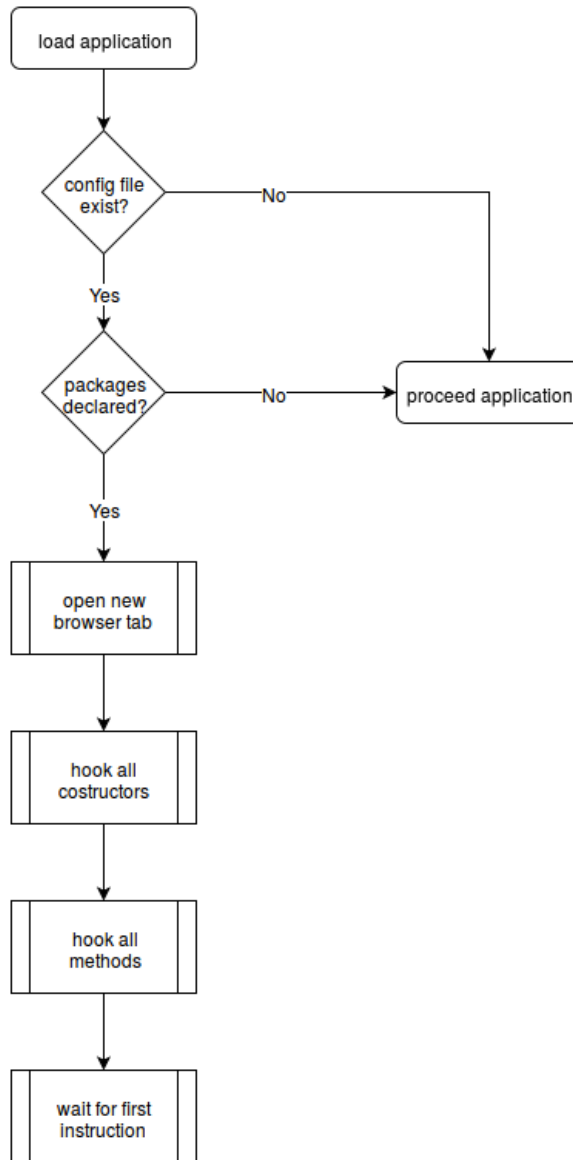
All information will be stored on a single file named *damn_packages.pkg* that is declared in the class *at.fhooe.mcm.faaat.XModule*. The structure of this file is very simple. We hold all package names as a String in an ArrayList which we write as an object into an *ObjectOutputStream*. On the other way around, when the investigated application gets loaded, it will read this file and take a look if any packages are declared in there. For a deeper look into the loading of an investigated application take a look on this section 7.6.

7.6 Flow of Loading a Tracked Application

Once the user choose an application to investigate and the configuration file is placed with the proper information, DAMN can read it and hook into it. This process is heavily bounded on Xposed and its way of functionality which is described here 6.3. If DAMN is installed and configured, every time an application gets loaded we check if the configuration file is available on the root directory of the application. If the file exists and is not empty, Xposed hook every package which is defined in this file. On this very early stage we already filter only defined packages. This gives us performance improvements as well as avoiding uninteresting classes. If the user decides later on to change the packages, the application has to get killed so that it has to get loaded again.

Before DAMN starts hooking classes, it calls the *newAppStarted* JNI function from the *damnserver.so*. This will open a new tab on the connected browser where the user can interact with the investigated application after hooking is finished. After this call, our Xposed module iterates through every declared class of the application and checks if the classes were defined in one of the packages declared in the configuration file 7.5. If this is the case, it will hook into all constructors and methods in this class.

Because of some performance improvements we parallelize the hooking process. Those performance improvements are also common on other applications which could lead to have multiple threads hooked with DAMN. It is possible to investigate each of those threads, but since most of this threads are performing small tasks and are dieing afterwards, we decided to not react on them by open a new browser tab. We only let the first thread communicate with the browser and ignore all other threads. It would not be a big deal to implement that every subprocess also gets a new tab on the browser, but it turned out that this is annoying and does not provide lots of advantages on most investigated applications.

Figure 7.9: Load Application Flow

7.7 DAMN Server Process

To make this all possible, we need another component which is running on Android as a daemon process. This process makes use of the shared library *libdamn-server.so* and the executable *damn-server-exec*. The executable will be triggered by the init process that makes it possible to start it at a very early stage of the boot process of Android. It makes use of the shared library to start the Civetweb server that can be accessed at this time.

7.7.1 Communication

As this part of DAMN running before the Android environment is set up completely, it allows also to investigate system applications or services. The communication to interact with this library uses JNI. All implementations done in Java can use the JNI interface which hides the complexity of used IPCs as pipes.

This server runs in its own process and is isolated from the memory of other processes like applications. Because of this it is not possible to use simple function calls from the library to communicate between those process. To make this possible, we are using a Linux IPC mechanism called pipes. They make it possible to communicate between the server process and the investigated application. Communication to the browser interface is implemented in the server process which makes additional IPCs obsolete.

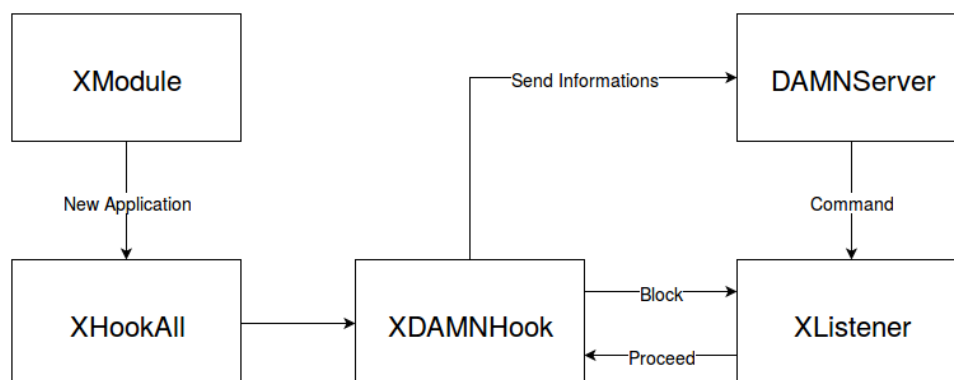
7.7.2 Document Directory

This server process is a customized Civetweb server. For the communication over HTTPS it also needs files which will be displayed on the browser. Those files are stored in the document root directory that consists of the HTML pages and some images which are used for the representation.

7.8 DAMN Xposed Module

On the previous section we described DAMNs flow of loading an tracked application. Figure 7.10 shows the flow of DAMNs Xposed module during investigation of an application.

Figure 7.10: Runtime Flow



7.8.1 Hook Process

Let us look closer to the classes we have implemented. *XModule* is the first class that will be informed if a new application gets loaded. It is our defined Xposed module and checks if the loaded application is a tracked application of DAMN. In that case the *XHookAll* class hooks into every constructor and method of the classed declared in the packages which are defined in the configuration file 7.5.

7.8.2 Control Flow Architecture

The whole control process of DAMN is implemented in *XDAMNHook* and a little helper class *XListener*. Both classes are communicating with the *DAMNServer* which provides a JNI interface for the communication with the DAMN server that is running as a separated process 7.7. This makes the complex structure of IPC communication abstracter and simpler to use.

The *XDAMNHook* class is controlling the application which is tracked by DAMN. It listens for commands from the browser interface and handles them. Furthermore it can stop the application at any given hooked method as well as manipulate values of it. This is realized with Xposed and Java Reflection features.

In the next section we handle the different states which investigated applications can have.

7.9 DAMN Runtime States

After loading and hooking all constructors and methods, we can interact on the browser with the running application and put it into different states. The very first state will be *pause* to bring the opportunity to step through the application at the very beginning. Here are all steps listed:

- Run
- Pause
- Step

7.9.1 Run

If we put the application on this state, we can interact with it on the device like without having the investigation turned on. Except with a little performance overhead, we should not face any difference on the behavior.

7.9.2 Pause

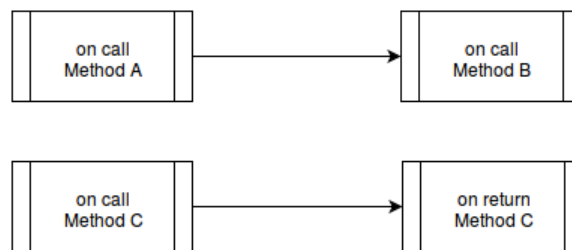
As written above, this is the first state on which the investigated application is put on. In this state, the application is freezing completely and will not

perform any calculations except for subprocesses. Any touch or key inputs will be ignored until we set the state to *run* or *step* again. It is also recommended to set the display timeout to a higher value preventing the device to change into a locked screen while the application is in this state.

7.9.3 Step

With the stepping feature we have the opportunity to use this tool almost like a debugger. But since we are not manipulating the source code, we only can interact on the method level. This means that if we take a step, we only can step from one method call to another or to the returning from a method. A short example should bring clarity 7.11.

Figure 7.11: Possible Interaction Points



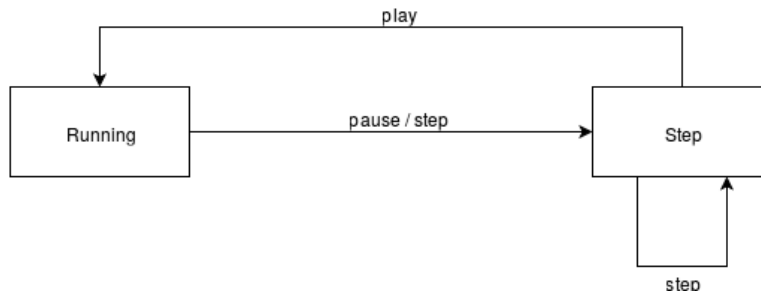
We can see two examples on this figure. Let us assume that the application is on *method A* and the user switch into the *step* state. As soon as *method A* calls *method B*, the application will block on the entry point of *method B*. The very same would happen if we switch into *pause*.

Another way would be in the second case, if the user switch into *step* state while the application is in *method C*, it will also block if *method C* will return. So basically, we have two interaction points where we can pause the application. The first is the call of a method and the second is the returning of a method.

On real applications this can be very nested since a method can call different methods which also may call further methods and so on. Therefore we can not expect that the next step will give us the return value of the actual method.

7.9.4 States of Investigated Application

To sum up this section, we show the possible states on this figure 7.12. As previously mentioned, if the user switches into *pause* state it will technically not differ from switching into the *step* state.

Figure 7.12: Application States

7.9.5 Obfuscated Applications

We talked earlier about obfuscation 7.5 and how it influences the way of investigation. DAMN is developed with obfuscation in mind and provides a different way of investigating such obfuscated applications. The current state of doing this is to decompile and investigate the obfuscated code. This takes quiet a long time since the code can be cut up into many different parts and searching for the interesting parts take its toll. With DAMN we are able to start the application, switch it back again to *play* and navigate to a section where such interesting things happen. If the user is in such a position, he can switch into the *stepping* mode and first have the possibility to look where he is on the code level. This already can give a clue in which classes we could find interesting parts. The next feature it offers is the stepping through those methods which shows pretty fast how the methods are interacting with each other without take a deeper look into the code itself.

This was the original purpose of DAMN but it does not stop here. The next section will give more information about an additional feature which can be used to directly test some cases of behavior manipulated values.

7.10 Manipulate the Application

Since now we discussed how we can interact on an application, we want to give an overview of another feature of DAMN. Because we are using Xposed for the dynamic interaction with the running application, we can also profit from the feature of manipulating values. We already gave an introduction into Xposed and its features and will now explain how it is used in DAMN.

7.10.1 Manipulation of Parameters

The first opportunity we want to explain is the manipulation of the parameters which are passed to a method on calling them. If the application we want to investigate is paused, we can look into the passed parameters on the

browser page. This is shown as a simple *JSON*(JavaScript Object Notation³) format like this:

Listing 7.2: Snipped of Parameters from the Browser Interface

```

1 {
2   "class": "net.cactii.flash2.TorchService",
3   "method": "onStartCommand",
4   "parameters": [
5     {
6       "2: Integer": 0,
7       "3: Integer": 1,
8       "1: class": "android.content.Intent"
9     }
10  ]
11 }
```

As we can see, it is a simple JSON structure we used to display the values. DAMN always will send three key-value pairs, the *class*, *method* and the *parameter*. Class and method values give detailed information about which method got called. In the parameter we have a nested JSON object that give more information about the passed parameters. The keys will gives details about the position and class. The value of the JSON entry will either be the value of the class or the class as a string if it is not supported. If there are no parameters for that method, it will be empty.

If it is a supported class, the value can be manipulated. In the example above, we could change the second parameter from zero to one. If we would change the first parameter it will be ignored because the Intent class is not supported.

7.10.2 Manipulation of Return Value

On the return of a method we also have the opportunity to manipulate the values. Since Java has one return value maximal this is simpler. Beside this value, we would also be able to change any other value of the application at this time. If the method that returns is of type void, we can not change the return value because it does not exist. Here a short snippet how this looks on DAMN:

Listing 7.3: Snipped of Return Value from the Browser Interface

```

1 {
2   "result": {},
3   "class": "net.cactii.flash2.TorchService",
4   "method": "net.cactii.flash2.TorchService"
5 }
```

In this particular example above, we have a void method that will pass no return value which we can see if we look at the key-value pair *result*.

³<http://www.json.org/>

7.10.3 Manipulatable Classes

That's it in terms of manipulation of an investigated application. Currently, only some basic classes are supported. But this can be easily expanded in the future. If we think about the possibility to strip every class into its essential parts, it would also be possible to alter custom classes. Here is a list of all supported classes:

- String
- Integer
- Long
- Boolean
- Float
- Double
- Character
- Byte
- int
- long
- boolean
- float
- double
- char
- byte
- byte array

7.11 Behavior Rules

The possibility to stop the application or resume it as well as the feature to change values in parameters or the return values makes it handy to use certain rules to automate some of those actions. Such rules would become very interesting on multi-threading where we also want to control the behavior of single threads. Another operational area is to automate recurring tasks as the one we will see in chapter 8 where we manipulate the value of a parameter which gets called iteratively.

7.11.1 Structure

The basic idea behind these rules is to have a certain trigger where we can react on. A trigger activates a rule which is then performed with its defined actions. Those triggers and actions can also be combined with *AND* or an *OR* operator to define more complex rules.

7.11.2 Triggers

Before we can use rules and make use of actions we need a trigger. Here is a list of triggers that can be useful:

- Method
- Value
- History

Method

A method can be a trigger if it gets called or is returning because we always have those two possibilities. To use a method, we also have to declare its class because a method name is not unique.

Value

Values can also be a trigger but only make sense if we combine it with the method trigger. Otherwise it would be called every time a parameter or return value has a specific class and value. As a value could belong to more than one possible class it make sense to define the class too.

History

Methods can be called from multiple other methods and it is not always interesting from every of them. With the use of a history trigger we can declare a method that is calling our defined method where the rule is applying. That helps to sort out calls of unimportant methods and get a cleaner output.

7.11.3 Actions

Rules would be useless if we would not have some actions which they could perform. The following list and describe them afterwards:

- Manipulate
- Log
- Pause

Manipulate

As the name indicates, this action can manipulate the value of either a field, parameter or return value. This is very useful if we want to alter the behavior of a loop since we do not want to change it manually on every iteration.

Log

Sometimes it is also interesting to get information at which time something is called or returning. This can be either achieved by single stepping through all methods or, a more elegant way, to define a rule with the log action and let the application run on the device. The logs will be written into the Android *logcat*⁴.

Pause

Since DAMN is kind of a debugger, we also need the opportunity to set a break point where the application will stop if it is reached. This can be achieved with the pause action. In combination with the history or value trigger it makes it to a more fine grained break point mechanism as on most other debuggers.

7.11.4 Chaining Triggers or Actions

The opportunity to combine one or more triggers as well as actions in one single rule makes it possible to define complex rules if needed. That can be very selective and helps to test quickly what impact some changes of parameters have on a running application without doing this very time manually.

7.11.5 Current State

Currently those rule base approach is not implemented in our release. Since it is a nice way to automate some interactions it will be added in the future 9.

7.12 Web Socket Data Exchange Protocol

DAMN uses a browser to make the interaction easier. The communication between the browser and the Civetweb server is handled with a WSS connection. Different information will be transported with different codes that the browser can interpret. Let us take a closer look how this information will be sent and interpreted.

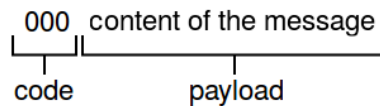
7.12.1 Protocol Structure

Before we explain the different codes which are used in the protocol on the different pages, we will take a closer look on how its structured. Because we do not have a lot of different commands, we did not have to use a very complex protocol. We decided to use the first three characters to distinguish between commands. The rest of the message is the payload which differs

⁴<http://developer.android.com/tools/help/logcat.html>

from command to command and has a variable length. In figure 7.13 we can see the structure of a message of our protocol in more detail.

Figure 7.13: DAMN Web Socket Protocol Structure



We show the different *codes* in the next sections below. They are separated by the two pages we have, the Start and the Tracking page.

7.12.2 Protocol Codes for Start Page

The start page is pretty simple as it is only shown if the device is connected through a web socket or not. Nevertheless, it also opens new browser tabs if an application will be opened which gets tracked by DAMN and this will be triggered by a command.

Code	Action
000	open new browser tab

As implied above, the one command on this page is to open a new *tracking* page. This is done if the code is *000* and this will trigger the following command:

Listing 7.4: JavaScript Command on Code 000

```
1 window.open('https://' + window.location.host + '/track.html?' + data);
```

The data from the command is used to give the *tracking* page the information to which application and thread it should connect to. This information will be handled in the Civetweb implementation and can handle multiple socket connections.

7.12.3 Protocol Codes for Tracking Page

On the *tracking* page there are more codes we have to handle. This page is used to show information about the tracked application and send commands to interact with it.

Code	Action
001	play
002	pause
003	step
004	source code
005	global variables
006	parameters
007	return value
012	close connection

The first three codes will be sent from the browser to the device and not the other way around. They are control commands to interact with the tracked application. Another difference between the other controls are that they will not use any payload since this is not necessary as the code itself is unique.

All other codes are used to get more information about the application onto the Tracking page. If the application got decompiled from our script 8.5 the source code of the actual class will be sent with the code 004. This helps the user to get more information about the class and the method which was called or is returning. The same applies to the code 005 which shows the actual defined global variables and fields. If a method gets called, its parameters will be assigned and displayed with the code 006. Same if the method returns, it will displayed the return value with 007 if the method has one. The last defined code is 012, which close the socket connection and the tab itself. This gets interesting if the multi thread handling is activated because a lot of tabs will be opened for some small operations and therefore can be closed automatically if they end.

7.13 Summary

In this chapter we described in detail how we are using different tools and system mechanisms in DAMN. It should be now clear how this tool is structured and how it works in Android. To get some further knowledge about how to use this tool, we provide some sample investigations with this tool that can be found in the next section.

Chapter 8

Investigating Real World Applications

So far we declared how DAMN works and what we can handle with it. It is time to use it on some applications to get a deeper understanding of its functionalities. But before we start with investigating applications, we want to give detailed information about the test environment we used.

8.1 Test Environment

Before we start to make some investigations we want to make it possible to rebuild the test environment to make it possible to everyone to try DAMN. We listed all equipment and software we used for this project.

8.1.1 Hardware

To rebuild our test environment it should be enough to use the same software. But here is the short list what hardware we used too:

- HTC M7
- Samsung N900X

8.1.2 Software Used on Device

More interesting is the listing of the software we are using for our components. Let us begin with the list of software on the Android device:

- CyanogenMod 11-20140106-NIGHTLY-m7ul
 - Android version 4.4.2
 - Kernel version 3.4.10-CM-gaf19676
 - SELinux Enforcing
- SuperSu version 2.46

- Xposed
 - app_process version 58
 - XposedBridge.jar version 54

8.1.3 Software Used on Computer

In addition the software that we use on the computer where only the browser is actually relevant:

- Linux Mint with KDE x64
 - Firefox version 42.0
 - Eclipse Mars.1 Release (4.5.1)
- Android SDK Tools 24.4
- Android SDK Platform 23.0.1
- Android NDK r10e

8.1.4 Used Tools

Beside this, we are also using the following tools with the latest available source from the repositories:

- d2j-dex2jar¹
- jadx²
- adb³
- ndk⁴

8.2 Setup

The complete source code of DAMN is freely available for non-commercial usage. It can be found here⁵. After cloning the repository and importing it into Eclipse, the IDE will automatically build the source code. In addition to this build, we also have to build the native sources. This can be done manually with the NDK command `ndk-build 3.10.2` or with a simple script we provide which also install it directly onto the device 8.1. This script has to be manually edited to work on every computer, since the paths will differ. We can also see, that the build commands are pretty simple- in this case the first two commands in the script. All other commands using the adb 3.10.1 to prepare the device and pushing the new libraries as well as one executable file onto it.

¹<https://github.com/pxb1988/dex2jar>

²<https://github.com/skylot/jadx>

³<http://developer.android.com/tools/help/adb.html>

⁴<http://developer.android.com/tools/sdk/ndk/index.html>

⁵<https://github.com/baer-dev1/DAMN>

Listing 8.1: Build and Push the JNI Parts of DAMN onto the Device

```

1 #!/bin/bash
2 ~/android-ndk-r10e/ndk-build clean
3 ~/android-ndk-r10e/ndk-build
4
5 adb wait-for-device
6 adb root
7 adb wait-for-device
8 adb remount
9 adb wait-for-device
10
11 adb shell 'rm -f /system/lib/libusb-tethering.so'
12 adb shell 'rm -f /system/lib/libcivetweb.so'
13 adb shell 'rm -f /system/lib/libdamn-server.so'
14 adb shell 'rm -f /system/lib/libdaemonize.so'
15
16 adb push ~/workspace/Eclipse/master/faaaat/obj/local/armeabi/libusb-
  tethering.so /system/lib
17 adb push ~/workspace/Eclipse/master/faaaat/obj/local/armeabi/libcivetweb
  .so /system/lib
18 adb push ~/workspace/Eclipse/master/faaaat/obj/local/armeabi/libdamn-
  server.so /system/lib
19 adb push ~/workspace/Eclipse/master/faaaat/obj/local/armeabi/
  libdaemonize.so /system/lib
20 adb push ~/workspace/Eclipse/master/faaaat/obj/local/armeabi/damn-server
  -exec /system/bin

```

We can now already run the application from *Eclipse*⁶ which pushes it onto the device and will start it the first time. The application will also put the root directory of our Civetweb server in its installation directory. Xposed should also have notified us already about a new module that is available and will get active on reboot. But first we have to do some more manual work.

To get this server started during the boot of Android, we will need to write this little script onto the device 8.3. We have to place it under the directory */system/etc/init.d* and can give the name *87damn*. While the number is the order in which it will get started by the init process, the name afterwards can be anything.

Listing 8.2: Simple Start Script

```

1 #!/system/bin/sh
2 /system/bin/damn-server-exec /data/data/at.fhooe.mcm.faaat/files/root/
  docroot /data/data/at.fhooe.mcm.faaat/files/root/ssl_cert.pem /data
  /local/tmp &

```

The only thing this script does is to run the *damn-server-exec* program with three parameters. The first one sets the root directory of the Civetweb documents. The second one tells the program where he can find the SSL

⁶<https://eclipse.org/>

certificate for the HTTPS connection and the last one is used for temporary files like IPC pipes.

If we now list this directory we should see something similar like this:

Listing 8.3: Simple Start Script

```
1 root@m7ul:/system/etc/init.d # ls
2 00banner
3 50selinuxrelabel
4 87damn
5 90userinit
6 99SuperSUDaemon
7 root@m7ul:/system/etc/init.d #
```

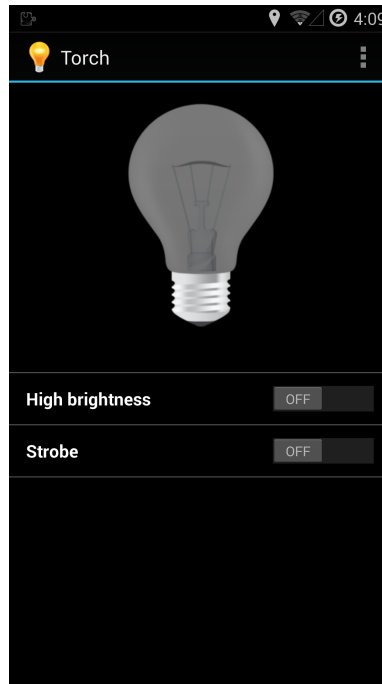
Here we can also see the *00banner* which is called in a very early stage (can be seen if we look at the logcat of the device from the boot process on) as well as the *SuperSUDaemon* which also gets started with this mechanism. We already explained SuperSU here 6.1 and showed that it is using also other techniques to start on boot process. This method may not work on every device and therefore has to be adapted individually. If someone have trouble using this solution on his own device, take a look at SuperSU and its solutions 6.1.

After a reboot everything should work as expected. This can be tested if we try to connect to the device in the browser. To get the IP address of the device we have to take a look at the settings under *About phone -> Status -> IP address*. Put this address into the browser and the start page should be displayed.

8.3 Simple System Application

After setting up the environment for DAMN we will test the functionalities on a simple application. On CyanogenMod there is a little application pre-installed named Torch which we will use for this purpose. Beside two options, the main functionality this application has is to turn on and off the flash light of the device by touching the light bulb icon on the screen which this figure 8.1 shows. But before we can investigate it, we have to set the configuration in the DAMN application. So let us start DAMN and choose the *Torch* application which will open the detail activity. In this case we only see one package called *net.cactii.flash2*. Tick this package and press the *Track* button. Now it should look like this figure 8.2.

One very handy feature on *CyanogenMod* is hidden in the *Developer options*, called *Kill app back button*. If we took a look at the Torch application before we tick it in DAMN, we have first to kill it in case we started it previously. This is important because our Xposed module will need to get loaded with the application we want to investigate. If this is not the case, we can not manipulate it. That can be done by open the application and long

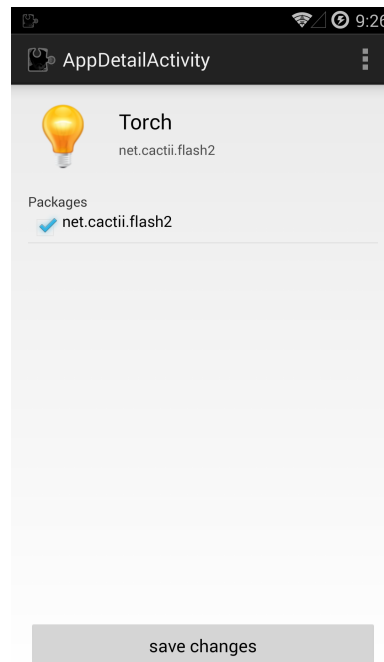
Figure 8.1: CyanogenMod Torch Application

press the back button. After a short amount of time it will be closed and a short message will shown up that it was killed. This feature gives us the quick opportunity to kill an already loaded application without restarting the whole system.

The next step is to load the Torch application. But before we do that, let us open the browser on the computer and connect to the start page of our device. It should print *Waiting for Applications* which indicates that we are ready to open the Torch. On open the application on the device, a new tab will be created on the browser. The first few seconds it will be blank, this is where Xposed hooks all methods from the chosen packages. After that we can see a rather black screen on the device and some information on the browser 8.3. The most important information is the JSON object on the *parameter* section which should look like this:

Listing 8.4: Frist Parameters of Torch

```
1 {
2   "method": "net.cactii.flash2.MainActivity$1",
3   "parameters": [
4     {
5       "1: class": "net.cactii.flash2.MainActivity"
6     }
7   ]
8 }
```

Figure 8.2: Torch in DAMN Detail Activity

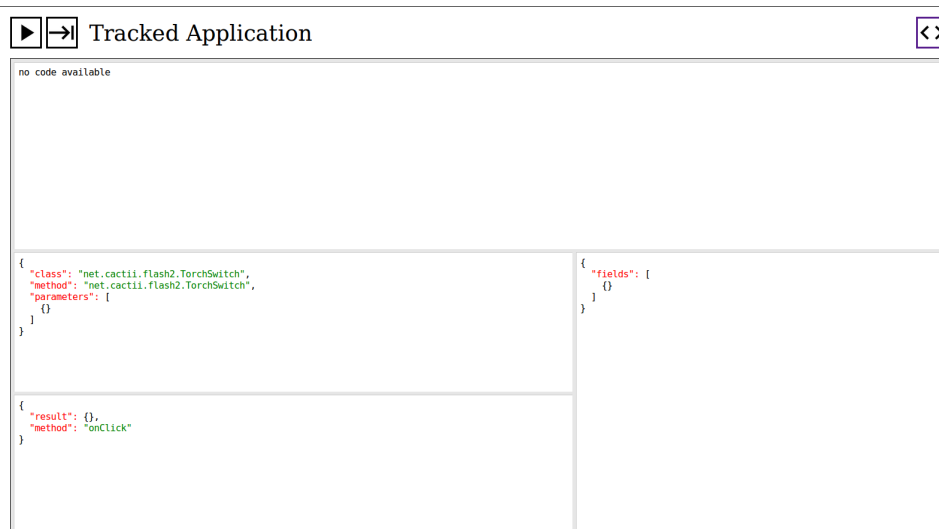
We can see that the first method which gets called on this application is the *MainActivity* in the package *net.cactii.flash2*. The only parameter which gets passed has the same class as where the method is declared in. We can also see that there are no fields declared and no source code is available. If we now press the play button in the browser, the application should continue as normal. Now we can also see the last return values in the *return* section. By pressing the light bulb image on the application, we can try if every thing works. The flash light should now simply turn on and off by pressing just like without investigating it.

Let us *debug* this application with DAMN. Be sure that the flash light is turned off and press *pause* in the browser. The only thing which should change is the icon of the pressed image on the browser. Nothing else should happen. Now we have to press the light bulb image again. What happens is, that we now provide an entry point for DAMN by reacting to a call which we triggered by pressing. In this particular case we can see in the browser, that the *onClick* method gets called. Nothing special happens here, so let us proceed by pressing *step* on the browser. We can recognize that this method will return without any value (seems to be a void method). Step again. Now we should see that the image has changed into an *active* state but the flash light is still off. Let us continue stepping till we can see that the flash light turn on. This should happen when the method *setFlashMode* with a integer

parameter is called. This integer with the value one will tell the flash light to turn on. Step over it and it should now shine. Since this application will call this method periodically we want to try to step again until this method gets called. But this time we will test the functionality to manipulate a parameter by editing the integer in the *parameter* section and change the value from *one* into a *zero*. By pressing again the step button, we should now see that the flash light is turned off without changed something on the device. That was the first succeeded manipulation of an application with DAMN. Let us proceed the application by pressing the *play* button again and stop this application.

This was a very basic example, but it shows how powerful DAMN can be. We did not even had the source code of this application but we easily found where the interesting parts are and also could manipulate them. We also could try to enter other values instead of zero and one and look what happens but for a simple example it should be enough.

Figure 8.3: Torch Application in Browser



But why did we not see any source code? Well, to be able to look at the source code of the application, we first have to decompile it. Unfortunately almost all decompiler use closed source libraries which are not compatible with the ARM architecture. If we also want to see the source code in the browser, we have to do a little manual work on the computer. This can be achieved by using this script:

Listing 8.5: Decompiler Script apk2damn

```
1 #!/bin/bash
2
3 if [ $# -le 1 ]
4 then
5     echo -e "usage:\tapk2damn <path to apk> <path to app installation root
6         dir>"
7     exit 1
8 fi
9 adb wait-for-device
10 adb root
11 adb wait-for-device
12
13 adb pull $1 app.apk
14
15 mkdir out
16 jadx --deobf -d out app.apk
17
18 adb push out $2
19
20 rm -rf out out.jar app.apk
```

We have to be sure that every single tool we use in this script is available on the Linux environment. If this is the case, we could now test our script with the following commands on the shell where the first command is used to make our script runnable:

Listing 8.6: Run Decompiler Script

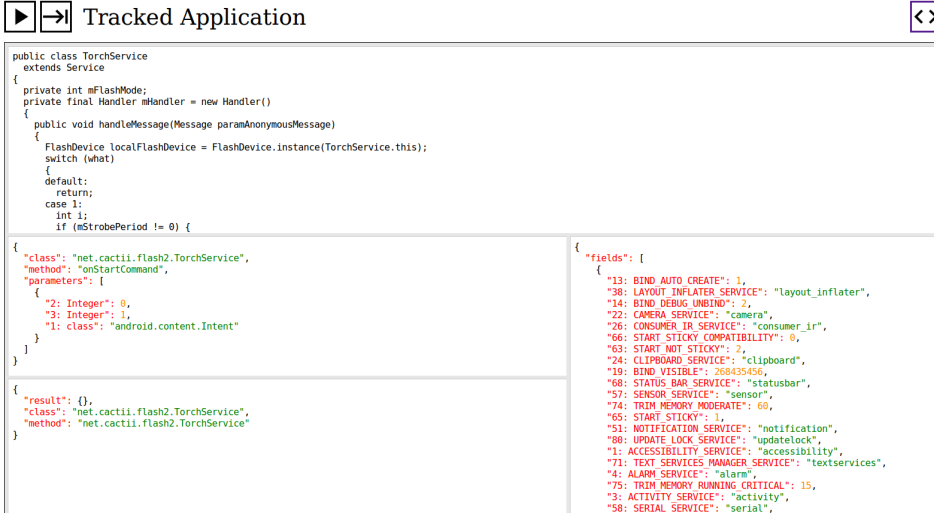
```
1 ~/tmp/ > chmod +x apk2damn
2 ~/tmp/ > ./apk2damn /system/app/Torch.apk /data/data/net.cactii.flash2/
```

Let us rerun our previous test with the Torch application and we should be able to see the decompiled source code in the browser as well 8.4. Now we have combined the advantages of dynamic analysis with the advantages of static analysis features successfully. With the available source code we can analyze a specific method in more detail because we can see what it is supposed to do.

8.4 Third Party Applications

While investigating such a small application like Torch is pretty simple, we also want to investigate two applications from the *Play Store*. Because we do not have the intent to blame a specific application we changed the naming of both applications we want to investigate. As quiz applications makes a lot of fun we decided to choose two different ones and take a deeper look into it.

Figure 8.4: Torch Application in Browser



```

public class TorchService
  extends Service
{
  private int mFlashMode;
  private final Handler mHandler = new Handler()
  {
    public void handleMessage(Message paramAnonymousMessage)
    {
      FlashDevice localFlashDevice = FlashDevice.instance(TorchService.this);
      switch (what)
      {
      default:
        return;
      case 1:
        int i;
        if (mStrobePeriod != 0) {

```

```

{
  "class": "net.cactii.flash2.TorchService",
  "method": "onStartCommand",
  "parameters": [
    {
      "2": Integer: 0,
      "3": Integer: 1,
      "1": class: "android.content.Intent"
    }
  ]
}

{
  "result": {},
  "class": "net.cactii.flash2.TorchService",
  "method": "net.cactii.flash2.TorchService"
}

```

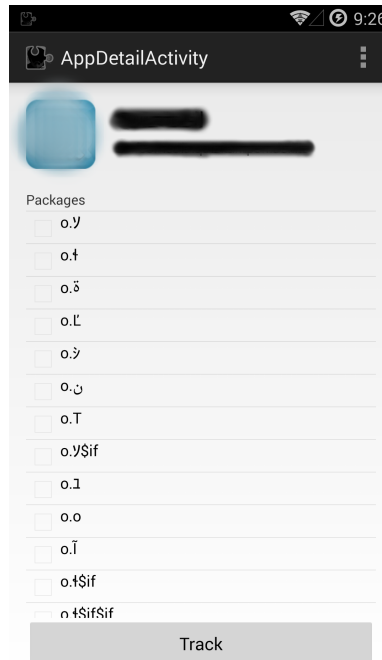
```

{
  "fields": [
    {
      "13: BIND_AUTO_CREATE": 1,
      "38: LAYOUT_INFLATER_SERVICE": "layout_inflater",
      "14: BIND_DEBUG_UNBIND": 2,
      "22: CAMERA_SERVICE": "camera",
      "26: CONSUMER_IR_SERVICE": "consumer_ir",
      "66: START_STICKY_COMPATIBILITY": 0,
      "63: START_NOT_STICKY": 2,
      "24: CLIPBOARD_SERVICE": "clipboard",
      "19: BIND_VISIBLE": 268435456,
      "68: STATUS_BAR_SERVICE": "statusbar",
      "57: SENSOR_SERVICE": "sensor",
      "74: TRIM_MEMORY_MODERATE": 68,
      "65: START_STICKY": 1,
      "51: NOTIFICATION_SERVICE": "notification",
      "80: UPDATE_LOCK_SERVICE": "updatelock",
      "1: ACCESSIBILITY_SERVICE": "accessibility",
      "71: TEXT_SERVICES_MANAGER_SERVICE": "textservices",
      "4: ALARM_SERVICE": "alarm",
      "75: TRIM_MEMORY_RUNNING_CRITICAL": 15,
      "3: ACTIVITY_SERVICE": "activity",
      "58: SERIAL_SERVICE": "serial",

```

8.4.1 Quiz Application A

Let us prepare DAMN for investigating the first application. For anonymity reasons we will call this application *Quiz A*. After installing it from the Play Store we first start the application to get familiar with it. After a while we can go to DAMN and open *Quiz A* in the detail view. First what we recognize is, that it takes quite some time until the view opens. The reason for that is easy to see if we look at the size of packages in the list as this figure shows 8.5. Here we can see very strong *obfuscation* in action. It has almost 200.000 constructors and methods which DAMN hooks. This huge amount of hooks can cause problems. First of all, we could run into low memory which would slow down everything and may crash the whole application. Luckily this is not the case here. The second problem is also a performance issue. Because of the huge amount of hooks which always gets processed by DAMN, it can slow down the application behavior quickly. Fortunately, we have a fast device where we do not feel much of this performance issue. A bigger problem could be that some of the classes we want to hook will not be hooked at all because they simply run out of time. Since there are some hooks which take a huge amount of time, we build in a time out. While this helps to hook smaller applications in a reasonable time, it can cause problems on very big one like *Quiz A*.

Figure 8.5: Quiz A Detail View

8.4.2 Investigate Quiz A

In a quiz application, the most important thing is the question and its possible answers. We want to investigate how the question gets processed. We want to clarify the following questions:

- How is the question loaded?
- How is the process of answering?
- Can we have influence on it?
- What can we do against this?

The reason why we want to clarify this question is, that we want to be sure that it is almost not possible to cheat on it. This is a very difficult problem in case of quiz applications because a very strong security mechanism could lead to weak usability.

Before we start, we will also run the *apk2damn* script to get all decompiled source code onto the device and therefore on the browser. Afterwards we can start right away to the first question.

How is the Question Loaded?

This should be one of the first question we should determine. It will give us the idea of how this system is constructed and how we can manipulate it.

Let us start the application as well as a new game. Before we try to answer the first question on the quiz, we press the *pause* button in the browser so that we can step through the single methods. Because of the huge amount of methods, this can take a while till we reach the interesting parts. But since we directly stepped into the point in the source where the first question is raised, it does not take too long.

After some calculations and passing of data about the device itself (Android version, device manufacturer, *IMEI*(International Mobile Equipment Identity)) we can see that some strings are passed which looks like a question and some possible answers. By stepping further, we can also detect which class will be used to store this information.

We also spotted that the questions are stored locally in a *sqlite*⁷ database. Since we have root access to the device, we can also investigate this databases directly. After some more investigations on the Shell where we use the `build` in `sqlite` program to read this databases, we figured out that there is a table where a lot of questions were stored on. But they are encrypted so we can not simply read it. Of course, we could look through the source where the questions get decrypted and use this information to decrypt all information in this database, but since we have directly access to the application we do not need this. As the application have to take care about decrypting the information before it will be displayed, we can simply use the information there.

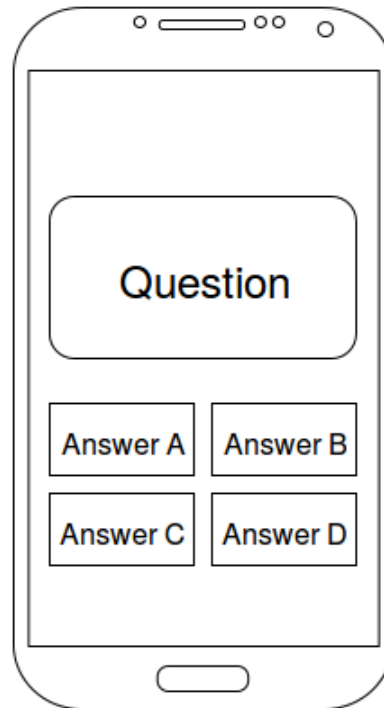
How is the Process of Answering?

After we have investigate how the question is loaded, we will check how the process of answering will be. Unfortunately we can not simply show a screen shot of *Quiz A* because anyone would immediately know which application it is. But for a better understanding we will describe how the displayed information will look and what possibilities a user have.

Every question we get will be displayed on top of the screen and we have four possible answers which we can choose by pressing them. This has to be done in a certain time or the question will be closed and we cannot answer it anymore. This is pretty much the same on many quiz applications and therefore expectable. Anyway, to get a better idea of it we made a abstract image of a simple quiz activity 8.6. This abstract picture also applies on quiz B.

The time for answering seems to get processed by a thread which we do not access (since multiple threads would be simple unproductive to investigate as described earlier 7.6). Luckily, if we click on one of the answer buttons, we will hook into the *onClick* event. From here we can see how the check mechanism works which decides if a answer is correct or not. This is very interesting on this application, because it seems to check if the chosen

⁷<https://www.sqlite.org/>

Figure 8.6: Abstract Quiz Question

answer is identical with the first possible answer from the database. That info combined with the knowledge that all possible answers will be randomly placed on the four buttons only leads to one conclusion, the right answer is always the first entry of the four possible answers stored in the database and on the object where the question is mapped on. After processing some more questions we can confirm that. This directly leads us to the next question we want to answer now.

Can We Have Influence on it?

Well, as we have the opportunity to manipulate the application as it is running on the device should already answer this question. Beside this, we have to answer how we influence it. There would be multiple ways of doing this. One could be to manipulate the timer and have infinite time to answer the question. While additional time could be useful, directly choose the right answer would be preferable.

To archive this, we can take a look at the method which check if the answer is correct. Let us take a look at a code snippet which does that:

Listing 8.7: Answer Processing on Quiz A

```
1    ...
2
3    Iterator localIterator2 = this.xU.iterator();
4    while (localIterator2.hasNext())
5    {
6        wR localwR4 = (wR)localIterator2.next();
7        if (localwR4.b)
8        {
9            if ((i3 != 0) || (j == 0))
10           {
11               int i4;
12               if (localwR4.a.b == wY.a)
13                   i4 = 1;
14               else
15                   i4 = 0;
16               if (i4 != 0)
17                   localwR4.setCorrect();
18               else
19                   localwR4.setWrong();
20           }
21       }
22   }
23   ...
24   ...
25
```

The first thing what everyone can perceives is that this is obfuscated source code which makes it really hard to understand. Thankfully DAMN made it easy to find this code section in a class out of almost 200.000 classes. On line twelve it gets decided if the chosen answer is correct or not. It simply tests if both values are equal.

If we now simply set *localwR4.a.b* to the same value as *wY.a* every time the *onClick* method in *o.xa* gets called, we choose the correct answer. This can either be done manually every time it is called with DAMN or we can write a Xposed module for this special case. Once it is loaded we can play against a competitor and be sure that we, no matter which answer we chose, get the correct one.

What Can We do Against This?

Now we know how easily we can find weaknesses in an application with DAMN, it is time to think what the developer could do against this. We saw that they already tried to add some more protection into the application because they encrypt the stored questions locally and use a very good obfuscation. Before we answer that question, let us try to investigate the other application in the next section.

8.4.3 Quiz Application B

The proceeding is pretty much the same as on *Quiz A*. We installed *Quiz B* from the *Play Store* onto the device and open DAMN to set the settings. As previously we should again ask questions which we should analyze during this investigation. To be comparable with *Quiz A* we use the same tasks:

- How is the question loaded?
- How is the process of answering?
- Can we have influence on it?
- What can we do against this?

Again, we use the apk2damn 8.1 script to decompile the application to directly investigate the source code in the browser. The structure of the displayed question is pretty the same as in *Quiz A*. There is one question phrase and four possible answers which can be pressed. After a short adoption time to get used to *Quiz B* we are ready to clarify the first question.

How is the Question Loaded?

The proceeding is the very same as on the other applications which we want to investigate. We connect a browser with DAMN and press *play* after hooking all methods. Now we navigate to a new game against a random player and press *pause* on beginning of the game.

While we step through the application we could see how *Quiz B* is using the *GCM*(Google Cloud Messenger)⁸ to communicate with a server to get new questions. It also uses a *GSON*⁹ class to extract JSON into Java objects, and vice versa¹⁰. In differ to the first quiz application which stored a few questions locally, this application will load every question from the server.

Another difference is that the application does not receive the right answer in this step. Therefore we can not extract the information which of the possible answers is correct.

How is the Process of Answering?

As on *Quiz A*, we can press one of the four possible answers to answer the question. Of course, on pressing one of the answers the *onClick* method is raised. This method figures out how long it took to answer the question in milliseconds and which of the four answers was chosen. After that, it calls the *answerQuestion* method which is shown in the following listing:

⁸<https://developers.google.com/cloud-messaging/gcm>

⁹<https://github.com/google/gson>

¹⁰<https://github.com/google/gson>

Listing 8.8: Answer Processing on Quiz B

```

1   ...
2
3   private void answerQuestion(int paramInt, long paramLong)
4   {
5       if (getActivity() == null)
6   return;
7       showProgress();
8       ApiRequests localApiRequests = (ApiRequests)ApiHelper.
generateBuilder(getActivity()).create(ApiRequests.class);
9       AnswerRequest localAnswerRequest = new AnswerRequest(paramInt,
paramLong);
10      localApiRequests.answerQuestion(this.mGameId, this.mQuestionIndex,
localAnswerRequest, new Callback()
11      {
12  public void failure(RetrofitError paramAnonymousRetrofitError)
13  {
14      QuestionFragment.this.hideProgress();
15      QuestionFragment.this.showError(paramAnonymousRetrofitError);
16  }
17
18  public void success(GameResponse paramAnonymousGameResponse, Response
paramAnonymousResponse)
19  {
20      QuestionFragment.this.hideProgress();
21      QuestionFragment.this.showCorrectAnswer(paramAnonymousGameResponse);
22  }
23      });
24  }
25
26  ...
27

```

Beside the process of answering the question, we can also see that the source code is much more readable as on the previous application. They did not use obfuscation and by looking at the logcat, they did not even turn debugging off.

This method shows how the application proceeds the answer of the user. As mentioned before, the integer value on the parameters is the index of the pressed button (0 - 3) and the long value is the time it took to answer. This is used to create a API request onto the server with the *gameId* and *questionIndex* which give some additional information about the actual game. An additional parameter is the a callback method which gets raised on server reply which either gives an error or the correct answer.

Can We Have Influence on it?

Before we do not receive the correct answer from the server, we have no option to get the correct answer from the application. This makes it much harder to manipulate it and give the correct answer right away like in *Quiz*

A. But we could try to change the time value to manipulate the timeout of answering the question. Here we can see the countdown implementation:

Listing 8.9: Countdown Quiz B

```
1    ...
2
3    private void countDown(long paramLong)
4    {
5        if (this.mCountDownRunning)
6        {
7            if (paramLong > -2L)
8                break label183;
9            this.mCountDownRunning = false;
10           answerQuestion(0, 12000L);
11           this.mViews.tvCountDown.setBackgroundResource(2130837635);
12        }
13        while (true)
14        {
15
16        ...
17
```

This gets called with the remaining time and can be easily manipulate by setting this value to a higher initial value. As a result, now we have time enough to do whatever helps us to answer the question correctly. The only thing what is left is to manipulate the answer time on the *answerQuestion* method. Let us change it to five seconds which is in the middle of the possible time range to answer the question.

Unfortunately, they also check at which time the application asked the server for a new question and the timeout on the server also closes the question nevertheless we change the answer time. We tried a few more tricks but the server implementation checked if the answer from the device is plausible and do not react on manipulated values.

What can We do Against This?

This question is a little bit obsolete here because they did a very good job. Although we had access to the source code and the possibility to manipulate the behavior we were not able to change the result.

But what about the obfuscation leak? As we can see in the reversed code, they did not use obfuscation but they where able to use other mechanisms to secure the application. Their design of how the data gets processed through the device and the server where correct and do not have any disadvantage of not using obfuscation.

And how to secure *Quiz A* finally? As they use another design as in this application it will be pretty hard to protect against such an attack. But the question should also be if it makes sense to secure a quiz application in a way that it is not manipulatable? Since we can not and moreover do not

answer such a question this has to be answered by a developer of such an application itself.

8.5 Recap

We investigated two quiz applications with DAMN and showed the result in this chapter. Our tool shows that the obfuscation which *Quiz A* was using was not strong enough to protect the application from being manipulated. *Quiz B* did not use obfuscation but it was designed proper and we did not have the chance that our manipulations have any impact.

This is only a short investigation but it confirms what we already assume, that obfuscation do not bring any additional protection to an application in terms of security. It only makes it harder to understand the source but dynamic manipulation tools like DAMN can bypass this.

Chapter 9

Future Work

During the implementation of DAMN we had faced a few problems we did not expect. Because we want to make it possible to investigate every application including system applications, DAMN operates on a very low level of the Android system where typical communication techniques between applications do not work. Fortunately we could rely on the Linux IPC features to solve this problems. Some of the described features are not fully implemented and others have the potential to get improved for a better usability. The next sections will pick some parts of DAMN and describe the future steps on them.

9.1 Behavior Rules

We described the idea of behavior rules that supports to automate some features of DAMN 7.11. This feature can be very powerful and the implementation is planned to be one of the next steps.

9.2 USB Tethering

Another feature we only described very shortly in this thesis was the communication over USB 7.2. We already implemented a native library with the inherent JNI interface to control this from the Java world. This feature can be triggered as well at a very early stage on the boot process because it does not rely on any Android specific components and gives the opportunity to investigate system applications that are started during the boot process. As most parts of this feature already are implemented we only have to add an option on the GUI to set this.

9.3 Multi Threading

Typical Android applications will have multiple threads to perform their purpose. Although DAMN supports multiple threads they will not be displayed in the browser because if every single thread will open a new tabulator it will be pretty unusable. We need some other way of displaying such threads that will not upset the user.

9.4 Stability

DAMN runs stable on most investigated applications. There are still problems in the hooking process from time to time which causes the tool to crash. It will need some further investigations to make it more stable.

9.5 Outlook

Our tool brings a new possibility to investigate applications whether they are obfuscated or not. As it is an open source project which is public available everyone can extend DAMN and bring new features into it. Other users of it may find further features or add more stability into this project. We look forward and hope a community of developers and we will actively improve this project further.

Chapter 10

Conclusion

We were developing DAMN with the purpose to combine static and dynamic analysis techniques into one tool to support the manual investigation of applications. The implemented tool provides a new way of investigating applications as it is possible to interact with it on the device while we can stop it at any given time. This helps to find code sections which are interesting in a reasonable of time. It can show its full potential on applications which uses strong obfuscation that have blown up the count of their classes and methods drastically. Furthermore, it can also manage other protection techniques an application can make use of as for example the protection of stored data through encryption. As the dynamic analysis parts of DAMN can investigate a running application, it is possible to read the values which are passed between method calls. As the data has to be encrypted at a certain point in code we can read those values directly. Manipulating those values are another feature DAMN provides and make it possible to test the runtime behavior on changed values.

Another statement DAMN can confirm is that security through obscurity is not guaranteed and should not be practiced. As DAMN provides a comfortable way of handling obfuscated applications it showed that the obfuscated code does not add security to an application. The same applies to Android applications which put some logical parts into native libraries to protect them for revering. Although this is not supported by DAMN it does nothing else than shifting the actual problem onto another level and do not provide a solution to fix it.

All those features makes DAMN a powerful investigation tool for analyzing Android applications.

Appendix A

Content of CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Files

Pfad: /

DAMN.pdf Master Thesis

A.2 Others-Files

Pfad: /stuff

android-distribution.ods Androids distribution pie chart document

A.3 Image-Files

Pfad: /images

android-arch.png illustration of Android architecture

android-distribution.png pie chart of distributed Android versions

damn-arch-v2.png illustration of DAMNs interaction components

damn-components-arch.png illustration of the layered structure of DAMN

damn-load-app.png . . . illustration of loading process

damn-states.png illustration of states of an investigated application

damn-stepping.png . . . illustration of DAMNs possible interaction points

damn-physical-view.png	illustration of the physical view of DAMNs components
damn-protocol.png . . .	illustration of DAMN WSS protocol
damn-xposed-arch.png .	illustration of Xposed structure in DAMN
quiz-abstract.png	illustration of a quiz application
screenshot-detail-browser.png	screenshot of DAMNs detail activity - browser
screenshot-list-system.png	screenshot of DAMNs list activity - system
screenshot-list-tracked.png	screenshot of DAMNs list activity - tracked
snapshot-browser-fields.png	snapshot of DAMNs tracking page separated
snapshot-browser-index.png	snapshot of DAMNs index page
snapshot-browser-track-torch-1.png	snapshot of DAMNs tracking page with Torch application - 1
snapshot-browser-track-torch-4.png	snapshot of DAMNs tracking page with Torch application - 2
screenshot-detail-torch.png	screenshot of DAMNs detail activity with Torch application
screenshot-obfuscated-packages.png	screenshot of DAMNs detail activity obfuscated packages
screenshot-torch.png . .	screenshot of Torch application
screenshot-xposed-modules.png	screenshot of Xposed Installer - modules

A.4 Implementation-Files

Pfad: /DAMN

assets	assets directory
jni	JNI directory
res	resource directory
src	source directory
AndroidManifest.xml . .	Androids Manifest file
build-android.sh	JNI build script
clean-android.sh	JNI clean script
ic_launcher-web.png . .	DAMNs launcher icon
lint.xml	lint file
proguard-project.txt . .	proguard config file
project.properties	property file

Special Terms

- ADB*** Android Debugging Bridge. 18
- API*** Application Programming Interface. 4
- ART*** ART is a runtime environment of Android. 12
- BASH*** Bourne-Again SHell. 42
- Binder*** is a Android specific IPC mechanism. 17
- C++*** is a programming language. 11
- Civetweb*** is a native open source web server. 42
- C*** is a programming language. 11
- DAC*** is an access control mechanism. 15
- Dalvik*** is a runtime environment of Android. 12
- FIFO*** is a named-pipe. 16
- IPC*** used to communicate between processes. 15
- JD-Core*** is a Java decompiler. 22
- JNI*** is used to communicate between Java and native code. 9
- JSON*** JavaScrip Object Notation. 61
- Java Virtual Machine*** runs Java applications. 9
- Java*** is a programming language. 9
- Linux*** is an operating system. 47
- MAC*** is an access control mechanism. 15
- NDK*** Native Development Kit. 18
- Play Store*** is the Android Application Market. 5
- Reflection*** gives information of a running Java application. 9
- SDK*** Software Development Kit. 18
- SELinux*** is a MAC access control mechanism. 15
- SuperSU*** manages privileged access on a rooted device. 37
- Unix*** is an operating system. 9
- Xposed*** is a dynamic manipulation tool. 39
- Zygote*** starts every Android application with a fork of itself. 18
- dex*** is a Dalvik executable file. 12
- dex*** is an optimized Dalvik executable file. 12

init is a process which starts other processes on boot. 11

jadx is a Android decompiler. 23

kernel is the fundamental part of Unix based operating system. 10

named-pipes is a IPC mechanism. 16

obfuscation is used to makes it hard to analyze reversed source code. 26

ptrace is a Unix system call which used to manipulate a process. 4

root is privileged access on a Linux system from the root user. 37

smali code is a disassembler code named after the the baksmali project. 4

socket is a IPC mechanism. 16

static analysis is the process of analyze source code. 32

References

Literature

- [1] Fabrice Bellard. “QEMU open source processor emulator”. In: *URL: <http://www.qemu.org>* (2007) (cit. on p. 5).
- [2] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O’Reilly Media, 2005. URL: <https://books.google.at/books?id=h0lltXyJ8aIC> (cit. on p. 17).
- [3] D. Chell et al. *The Mobile Application Hacker’s Handbook*. Wiley, 2015. URL: <http://books.google.de/books?id=5gVhBgAAQBAJ> (cit. on pp. 14, 20).
- [4] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997 (cit. on p. 27).
- [5] Joshua J. Drake et al. *Android Hacker’s Handbook*. 1st. Wiley Publishing, 2014 (cit. on pp. 5, 16, 32, 33).
- [6] Thomas Eder et al. “Ananas-a framework for analyzing android applications”. In: *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE. 2013, pp. 711–719 (cit. on p. 5).
- [7] David Ehringer. “The dalvik virtual machine architecture”. In: *Techn. report (March 2010)* 4 (2010) (cit. on p. 22).
- [8] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android’s Security Architecture*. No Starch Press, 2014. URL: <https://books.google.at/books?id=y11NBQAAQBAJ> (cit. on pp. 13, 15, 18).
- [9] William Enck et al. “A Study of Android Application Security.” In: *USENIX security symposium*. Vol. 2. 2011, p. 2 (cit. on p. 23).
- [10] William Enck et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5 (cit. on p. 3).
- [11] Pau Oliva Fora. “Beginners Guide to Reverse Engineering Android Apps”. In: *RSA Conference*. 2014 (cit. on pp. 21, 22).

- [12] A. Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Android Forensics: Investigation, Analysis, and Mobile Security for Google Android. Elsevier Science, 2011. URL: <https://books.google.at/books?id=i-yWIVd4z7MC> (cit. on p. 18).
- [13] Peter Hornyack et al. “These aren’t the droids you’re looking for: retrofitting android to protect data from imperious applications”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 639–652 (cit. on p. 3).
- [14] Martina Lindorfer et al. “ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors”. In: *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. 2014 (cit. on p. 5).
- [15] Robert Love et al. *Linux Kernel Development Second Edition*. Novell Press: Sams Publishing, 2005 (cit. on p. 10).
- [16] G. Nolan. *Decompiling Android*. SpringerLink : Bücher. Apress, 2012. URL: <https://books.google.at/books?id=WAAJEN4ZI-QC> (cit. on p. 28).
- [17] Damien Oceau, Somesh Jha, and Patrick McDaniel. “Retargeting Android applications to Java bytecode”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 6 (cit. on p. 23).
- [18] Pradeep Padala. “Playing with ptrace, Part I”. In: *Linux Journal* 2002.103 (2002), p. 5 (cit. on p. 4).
- [19] Nicholas J Percoco and Sean Schulte. “Adventures in bouncerland”. In: *Black Hat USA* (2012) (cit. on p. 5).
- [20] Y. Shi. *Virtual Machine Showdown: Stack Versus Registers*. Trinity College, 2007. URL: <https://books.google.at/books?id=41ZtMwEACAAJ> (cit. on p. 12).
- [21] Michael Spreitzenbarth et al. “Mobile-sandbox: having a deeper look into android applications”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM. 2013, pp. 1808–1815 (cit. on p. 4).
- [22] Christian Ullenboom. *Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6;[das umfassende Handbuch; aktuell zu Java 6; DVD-ROM inkl. Openbook-Bibliothek (4000 Seiten), 300 Aufgaben und Lösungen, Java 6 und Eclipse 3.2, viele Zusatztools]*. Galileo Press, 2006 (cit. on p. 9).
- [23] Vanessa Wang, Frank Salim, and Peter Moskovits. “Introduction to HTML5 WebSocket”. In: *The Definitive Guide to HTML5 WebSocket*. Springer, 2013, pp. 1–12 (cit. on p. 43).

- [24] Vanessa Wang, Frank Salim, and Peter Moskovits. *The definitive guide to HTML5 WebSocket*. Vol. 1. Springer, 2013 (cit. on p. 43).
- [25] Lukas Weichselbaum et al. “Andrubis: Android malware under the magnifying glass”. In: *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001* (2014) (cit. on p. 5).
- [26] Rubin Xu, Hassen Saïdi, and Ross Anderson. “Aurasium: Practical Policy Enforcement for Android Applications.” In: *USENIX Security Symposium*. 2012, pp. 539–552 (cit. on p. 4).
- [27] Lok-Kwong Yan and Heng Yin. “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.” In: *USENIX security symposium*. 2012, pp. 569–584 (cit. on p. 4).
- [28] Min Zheng, Mingshen Sun, and John Lui. “DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability”. In: *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. IEEE. 2014, pp. 128–133 (cit. on p. 4).

Online sources

- [29] Inc. IDC Research. *Smartphone OS Market Share, 2015 Q2 @ONLINE*. Nov. 2015. URL: www.idc.com/prodserv/smartphone-os-market-share.jsp (cit. on p. 7).
- [30] Jon Oberheide and Charlie Miller. *Smartphone OS Market Share, 2015 Q2 @ONLINE*. 2012. URL: <http://diyhpl.us/~bryan/papers2/security/android/summercon12-bouncer.pdf> (cit. on p. 5).