Runtime analysis of Android apps based on their behaviour

Eva Anker

TU Delft

Cover image from [1].

Runtime analysis of Android apps based on their behaviour

by

Eva Anker

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday January 9, 2020 at 10:30 AM.

Thesis committee:	Prof. dr. A.E. Zaidman,	TU Delft, chair
	Dr. ir. S.E. Verwer,	TU Delft, supervisor
	K.R. Valk, MSc	Riscure, company supervisor

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Abstract

In the modern world, Smartphones are everywhere and Android is the most used operating system. To protect these devices against malicious actions, the behaviour of Android apps needs to be studied.

Current tooling does not provide complete insight into the behaviour of an Android app. A tool was built to observe what goes on inside an Android app. The tool can hook all functions and change the outcome of a function call. It is possible to log all method calls to observe when a method is called, with their arguments and return values. Every call the app makes inside the JVM can be shown and a complete picture of the application can be obtained. During this process the app stays responsive and will not slow down significantly. The information provided can be used for building a call graph, finding vulnerabilities or checking for app detection mechanisms.

Preface

This thesis marks the end of my academic career. When I started as a freshman at the Delft University of Technology, I could not have imagined writing such a thesis. I gained more knowledge than I could have ever imagined. During my studies, I met many people and with quite a few, I became friends. These friends were there for me in the good and even more important during the bad days. I would like to thank them for all their support during my entire study and being there for me when I needed them.

Furthermore, I would like to thank everybody at Riscure for supporting me during my thesis and being very helpful. In particular, I want to thank my supervisor Kevin for all the support he has given me. He believed in me even when I doubted it, and I am grateful to have had such an amazing supervisor.

I would like to thank my university supervisor, Sicco, for making sure that I kept on track academically. The thesis project evolved in an engineering challenge and with his support it is still scientific enough. I want to thank Andy Zaidman for being the chair of my thesis committee.

A big thank you to my family for their support. Last, I would like to thank everybody that I did not thank personally, but still supported me during my journey.

I am excited to bring into practise what I have learned during all those years and to see what the future holds.

Eva Anker
Delft, November 2019

Contents

List of Figures

List of Tables

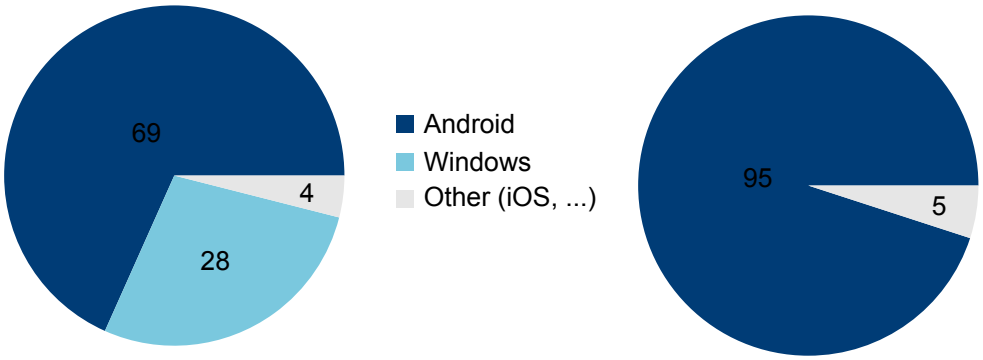
List of Algorithms

List of Listings

1 Introduction

In the modern world everybody has a smartphone and they are used more than desktops, tablets and consoles combined [5]. On mobile devices, Android is the most used operating system [6, 7].

Like any other device, viruses can infect mobile devices. The infection rate on the mobile network has been studied by Nokia [8]. A summary of the study is presented in Figure 1.1. On the left, all devices connected to the mobile network including Windows PCs are shown. On the right, only the invection rate of smartphones is shown. It can be seen that Android has the majority of infections, and Windows laptops are a second. From the smartphone only graph it can be observed that the percentage of infections is higher than the market share, respectively 95% versus 75-90% [6, 7]. This means that if you have an Android device, you have a relatively high change of getting infected, compared to the other mobile operating systems.



(a) Device infection rate (b) Smartphone infection rate

Figure 1.1: Infections in the mobile network for 2017

Mobile Android applications (apps) are mostly downloaded via the Google Play Store (play store). In 2017 Google removed over 700.000 apps from the play store, because they violated their policies [9]. However, one is not required to use the play store to download and install apps and other third party, stores can be used. In some countries not all functionality is available, which makes it impossible to buy some apps [10]. In China the usage of a third party store is even necessary as the play store is not available [11]. Finally, personal preference can also result in the usage of third party stores. These third party stores have different apps, and seemingly the same app could be from a different author. App owners seldom send their apps to multiple stores and their apps are copied and uploaded to other stores. It is almost impossible to upload to all stores due to their sheer amount.

Android stores can contain malicious content. Which are apps that exhibit malicious behaviour. Google defines this as apps that steal data, secretly monitor or harm users, or are otherwise malicious [12]. The term Potential Harmful App (PHA) is used by Google for malicious apps, where an app does not have to be harmful for all devices [13]. Apps that exhibit malicious behaviour as a request by the user, like rooting, are not a PHA. Third party stores contain more malicious content and it has been

shown that users that only download from the Google Play store are 8 times less likely to have a PHA on their phone [14].

The traditional way of defending against infections on PC is installing an antivirus scanner. Android virus scanners are not often installed. Android devices still have limited resources and an antivirus scan slows down the device. Only scanning on change after installing or upgrading would omit these problems. Hence, the Google Play Store has a function called Google Play Protection [15]. It is integrated with the Google Play Store on recent devices, and it has over 2 billion users [16]. Google Play Protect scans apps before and after installing for malicious content. It also includes regular checks on all apps on the device, to include the higher risk third party apps.

The protection provided by Google Play Protection is not as good as the other Android virus scanners. The German IT security institute AV-TEST have tested Google Play Protection and compared it to the industry standard [17]. During testing they found that the detection rate of Android malware in real time was 62.0% whilst the industry average is 97.8%. For malware discovered in the last weeks the result was slightly better with 69.2%, where the industry average is 98.2%. Google Play protection had a false positive rate of 7.7% whereas the industry average is 0.4%. This shows that the amount of true positives is relatively low, whilst the amount of false positives is relatively high.

We have seen that Google Play Protection does not detect all PHAs. It does not classify all intended harmful behaviour as malicious. These apps can undermine the security of Android by weakening or disabling security features. Android apps might contain banking information or intellectual property, which needs to be protected. These apps need to defend itself against malicious content or apps that weaken the security. An app can defend itself by using countermeasures, and there are two types of countermeasures. Firstly, there are countermeasures that detect unwanted behaviour, for example root or emulation detection. Secondly, there are countermeasures that protect data, such as white box cryptography.

Checks for both malicious content and countermeasures are important. On the one hand, the malicious content has to be found. On the other hand, it should be checked that apps defend itself against possible malicious content. To check both malicious content and countermeasures analysis on Android apps has to be done. It is important to use a generic approach, as malware changes over time and countermeasures hold different behaviour. Behavioural analysis of Android apps has to be done that is not geared towards one type of behaviour. This behavioural analysis could then be added to a tool like play protection, to increase the detection rate.

1.1. Cyber Security

Cyber Security is defined as the ”preservation of confidentiality, integrity and availability of information in the Cyberspace” by the ISO [18]. Whereas they define Cyberspace as the ” complex environment resulting from the interaction of people, software and services on the Internet by means of technology devices and networks connected to it, which does not exist in any physical form”. Following this definition it can be seen that confidentiality (C), integrity (I) and availability (A) are the key components for security, these are also called the CIA triad. In this context confidentiality means only authorised people can see and author data, integrity ensures that the data is unchanged and lastly availability ensures that the data it is accessible at all times [19]. In Figure 1.2 a graphical view of the CIA triad is shown.

Only if all three components are present full security can be obtained, and this is shown with the colour green. Without confidentiality the data can be read, without integrity data can be modified and without availability access to the data could be interrupted. A PHA will breach one or more of these components. To illustrate, rooting an Android phone could breach all these three components. Firstly, a user has access to more parts of the system where it could read data. Secondly, it is possible to write or modify data on the system. Lastly, access to data could be changed by a root user. To what extend these modifications can be done depends on the protection by the app. Furthermore, Android has SELinux in place which can limit rooting access. More on SELinux can be found in Section 2.2.2.

1.2. Research Question

To learn more about the behaviour of Android apps, the following research questions are composed and are the basis of this master thesis.

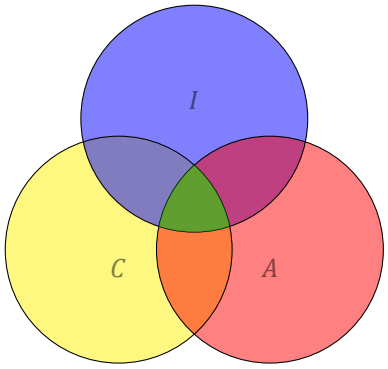


Figure 1.2: The CIA triad, where the green middle part is secure

RQ1: How can the internal behaviour of Android apps be analysed?

This research question aims to find an approach to do behavioural analysis on Android. The question can be split into two parts.

RQ1a: What current research is available that analyses the behaviour of an Android app?
Multiple approaches exists to do behavioural analysis on Android apps. In this research question the current research is investigated to find which techniques are used. Multiple techniques will be discussed in order to find the most suitable technique for the cause.

RQ1b: What tooling is available to conduct behavioural analysis with on Android apps?
Tools do not always have an accompanying paper. Tools for the techniques found in RQ1a are discussed and investigated. These tools are then evaluated to see if they can conduct behavioural analysis.

RQ2: How can a tool be build to hook all methods of an Android app to analyse its behaviour?

In research question one it will be shown that Dynamic Binary Instrumentation (DBI) is the preferred method for behavioural analysis. Method hooking is a type of DBI. In order for complete behavioural analysis, all methods of an Android app need to be hooked. This has not been shown in current research, and in this question it will be investigated if it is possible.

RQ2a: What is the efficiency of the newly created tool?
In this sub-question the tool will be evaluated and compared with currently available tooling.

1.3. Outline

In Chapter 2 context is given on the internal workings of Android and background knowledge for the rest of the thesis is given. In order to connect this thesis with related work, Chapter 3 explains the current state-of-the-art of Android reverse engineering. Afterwards, in Chapter 4 the currently available tools for Android DBI and their shortcomings are discussed. In Chapter 5 a new tool is proposed and the inner workings are explained. The tool itself is then evaluated and compared in Chapter 6. In Chapter 7 the project is concluded and future work is discussed.

2

Android

This chapter aims to explain the concepts of Android that are needed in the coming chapters. It is not mend to give a total overview of Android and should not be taken as such. Firstly, an introduction to the Android ecosystem is given. Secondly, some internal concepts of Android are discussed. Then the Java Native Interface and the Android Debugging Bridge are explained. Lastly, the difference of Java and Kotlin regarding this thesis is explored.

2.1. Android ecosystem

Android is an open-source mobile operating system, with a software stack build upon Linux. This stack is displayed in Figure 2.1 and it shows that Android consists of 6 distinct components. Each of these components will be explained below.

2.1.1. Linux kernel

In the core of Android there is a modified version of the Linux kernel. The responsibility of the kernel is the same as for other operating systems. It handles among other things the drivers, memory, resources and power management.

2.1.2. Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) sits on top of the Linux kernel and it is the bridge between the hardware and software. Other components will communicate with the HAL instead of the kernel itself. The HAL consists of multiple modules, each implementing an interface for a specific hardware component. The system will load these interfaces upon request of other parts of the software stack.

2.1.3. Android runtime

The Android Runtime component is designed to execute all Android apps and some system services. It consists both of the runtime itself and the core Java libraries. These libraries contain most functionality of Java, which is needed for the Java Application Programming Interface (API).

Besides these libraries, the Android Runtime itself is contained in this component. Java requires a Virtual Machine (VM) to execute its bytecode, however Android does not use the standard Java Virtual Machine. Android uses ART to function as its virtual machine and previously used the Dalvik Virtual Machine (DVM) otherwise known as Dalvik. The DVM has been replaced by Android Run Time (ART), although it is a runtime instead of a virtual machine. ART replaced a virtual machine and exhibits the needed functions, it is referred to as one.

The DVM could not translate Java bytecode to machine code and therefor dex files where introduced. This way Java bytecode is translated to Dalvik bytecode and stored in dex files. In order for Android apps to be backwards compatible the input format of ART and Dalvik are equal. Dex files are both stored inside system Java libraries and the Android Packages (APKs) itself.

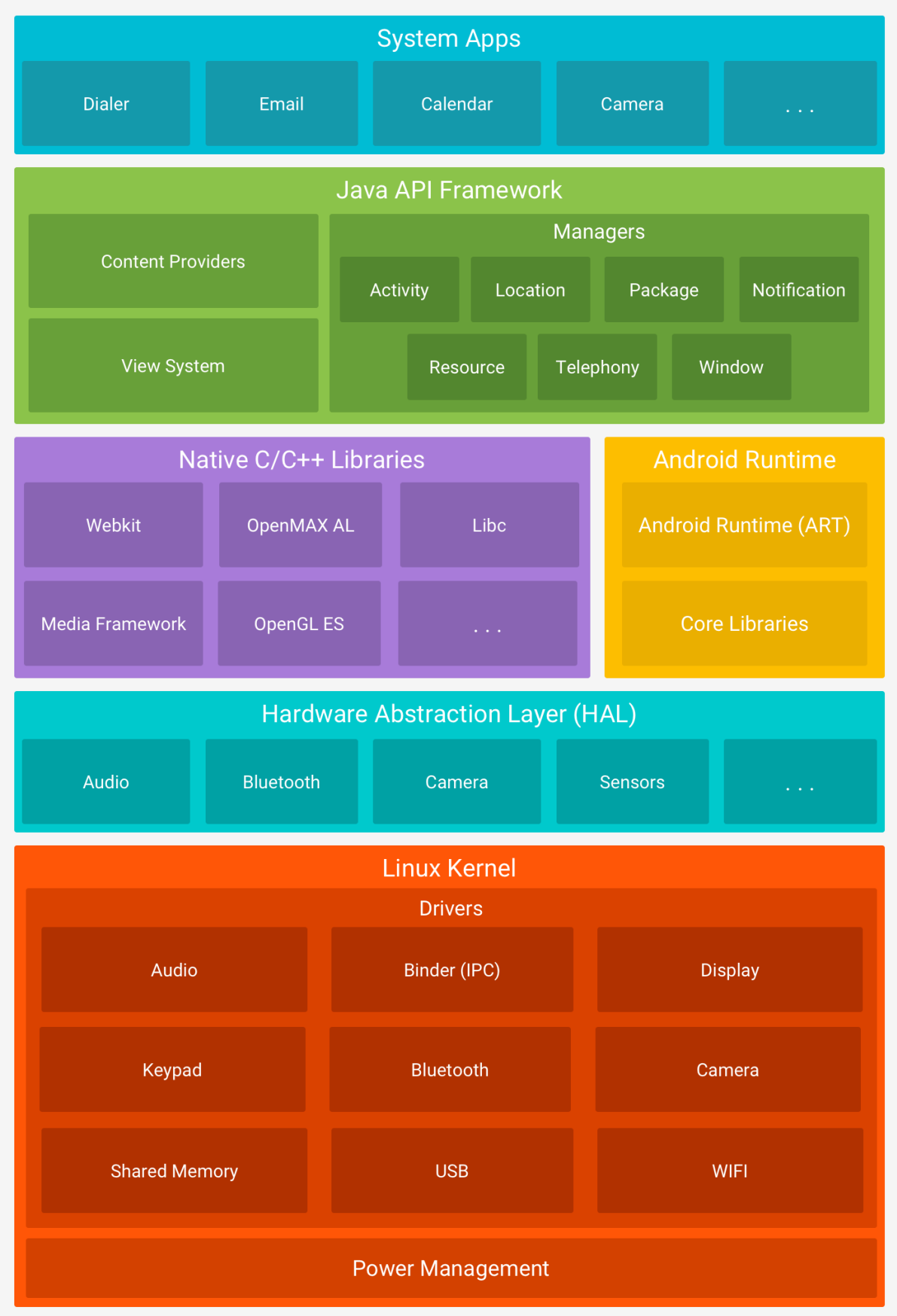


Figure 2.1: Android stack [2]

Differences between ART and Dalvik
Although ART and Dalvik accept the same input format, their internal processes are different. The DVM uses Just In Time (JIT) compilation, therefor during execution of an app dex code will be dynamically translated to machine code [20]. The longer an app is executed, the more code will be compiled and cached. Not all code will be converted, this will leave a relatively small memory footprint and will use a relatively low amount of physical space on the device. The DVM was designed for devices with limited resources [21, 22]. Over time, Android devices got more resources and ART was developed with less tight resource constraints. ART uses Ahead Of Time (AOT) compilation, which means that during the installation of an Android app the bytecode is translated into machine code and stored on the device. This way the installation of an app takes longer, however the execution will be faster. With Android 7, JIT support was added next to AOT. This enables code profiling capabilities that can constantly improve the performance of a running app [23].

Differences between ART/Dalvik and the standard JVM
Between DVM/ART and the standard Java Virtual Machine (JVM) the biggest difference is that the standard JVM is a stack based architecture whilst ART/DVM are register based architectures [24]. In a stack based architecture variables will be placed on the stack, whilst with a register based architecture variables are placed into registers. Register based architectures have 47% less instruction than stack based architectures, but the code is 25% larger [25]. The difference of stack and register based architectures will be shown with an example, in this example integer addition is done. On a stack this would take 3 or 4 instructions depending if a variable has to be pushed back on the stack. This is shown in Listing 2.1, where the fourth command is optional. With register based this takes 1 instruction, and the result is saved in a register, as seen in Listing 2.2.

```
1    pop   eax              ; Pop first argument from stack
2    pop   ebx              ; Pop second argument from stack
3    add   eax ebx          ; Add both arguments together to second argument
4    push  ebx              ; Push argument back onto the stack
```

Listing 2.1: Stack based integer addition

```
1    add r1 r2 r3           @ Add r2 and r3 to register r1
```

Listing 2.2: Register based integer addition

2.1.4. Native C/C++ libraries
This component holds native libraries written in C and C++ that have not been specifically designed for Android. The functionality of those libraries are exposed to other parts of the software stack. Some of these libraries, for example OpenGL are also exposed through the Java API to developers. From native code, some of these libraries can be directly accessed.

2.1.5. Java API framework
All Android features that are not included in a standard Java library can be obtained via the API. These features are grouped into three categories.

- **Content providers** The content provider manages data that is stored by both the application itself and other applications. If an application needs data from another application, the content manager provides a way to share this data securely. This only works if the target app provides a way to share the data via the content provider. The data shared via the content provider can both be set to read only and writable based on the preferences of the data owner.
- **View System** The view system can be used for building the User Interface (UI). It includes ready-to-use components that can be integrated into an application.
- **Managers** Inside the Java API framework there are multiple managers. Each manager, provides access to a specific part of the software. For example, by using the notification manager an application can have custom notifications in the status bar. An important and non-straightforward manager is the activity manager. An activity is a single screen with a user interface [26]. Every application consists of at least one activity. If allowed an application could be started from any activity from another application or on start-up. Invoking an activity of another application can be used for content sharing or adding attachments to emails.

2.1.6. System apps

Android devices contain two types of apps. Firstly, there are the system apps that come pre-installed on the device. These apps are needed to make full use of the device. Examples include sending emails, making calls or installing other applications. A second type of apps are the user apps. These are not shown in Figure 2.1 as they are not an essential part of the Android ecosystem. The functionality of system apps can be invoked from both other system and user apps. The difference between system and user apps lies in the fact how they are installed. It is difficult to remove system apps as this would remove some core functionality. In order to delete these apps measures have to be taken that might void the warranty of the device in question. User apps can be installed and removed from the device without problems. For user apps to obtain permissions the user has to grant them [1], whilst system apps have them by default.

Only the behaviour of applications is analysed during this work the other components are also important. An application would not be able to function without the other components. In this section a distinction is made between system and user apps, however on the runtime level this distinction cannot be observed as they both run in the same way. Only the term app is used that refers to both system and user apps.

2.2. Internal concepts

In this section some background knowledge is given over Android specific topics that are used in the coming chapters.

2.2.1. Zygote

In Android there is a special process that is forked for each new application, this process is called Zygote [27]. ART starts Zygote together with the first VM. The first VM calls the main method of Zygote to load all shared Java classes and resources [28]. Zygote itself contains a VM, and as each application is a child of Zygote it also contains a VM. Every application runs in its own VM, with its own thread. This way every application runs its own version of ART.

2.2.2. SELinux

Security-Enhanced Linux (SELinux) is a generic Linux Security Module [29]. Starting from Android 4.3, SELinux is included and was in full effect as of Android 5 [30, 31]. SELinux operates on the principle of default denial, so everything that is not explicitly allowed is denied. It can function in two different modes:

- In **permissive** mode a permission denial is not enforced, but only logged.
- In **enforcing** mode a permission denial is enforced and logged.

Android is standard in enforcing mode and this cannot be changed with user privileges. On some devices root access is sufficient, for others kernel level privileges are needed [32].

2.2.3. Root

Normally an Android user has limited privileges and control over an Android device. When a device is rooted, a user gains the highest privileges possible and has full control over the device. These privileges can be restricted with the usage of SELinux. There exist multiple methods for rooting Android devices and most rely on vulnerabilities within Android [33]. Having a rooted device has some risks, and some apps have countermeasures that prohibit the app from running on rooted devices. To combat these countermeasures, root cloaking apps exist that try to hide if a device is rooted [34].

2.2.4. Address Space Layout Randomisation

Like most other operating systems, Android uses a technique called Address Space Layout Randomization (ASLR). ASLR is a technique that hides the memory layout of a system, to prevent adversaries from exploiting memory vulnerabilities [35]. More specific, the location of modules have a randomised

[1] In older Android versions, only none or all permission could be accepted. Nowadays a user can specify per permission if it will be granted.

offset as well as some in-memory and kernel structures. ASLR is used to increase the difficulty in mounting attacks that require knowledge of the memory layout.

2.2.5. Execution environment

Android devices have two different execution environments. There is an execution environment for bytecode and one for machine code. Bytecode is executed by the interpreter, where machine code can be directly executed. Native functions always have accompanying machine code, whilst for Java methods this is not the case. For all Java methods there is bytecode, and for some machine code is compiled. Machine code can be generated by the compiler, when a method is invoked. An example of this is a static method for which the class is not yet initialised. This method is then compiled and the resulting machine code is saved for a next function call [36]. There are methods that are always executed by the interpreter. For example, methods from abstract classes as their implementation differs on the context [37].

2.3. Java Native Interface

The Java Native Interface (JNI) is a bidirectional interface provided by Java in order to interact with C/C++. This C/C++ code is referred to as native code. This is mostly used by Java invoking a native function, where the native function will return to Java. However, given a running native program it is possible to attach to a Java program.

2.3.1. Java Virtual Machine

Every Java program runs inside a Java Virtual Machine (JavaVM). The JNI uses the term JavaVM instead of JVM. The JavaVM is process specific. It is possible to have multiple JavaVMs per process, although on Android only one is used. Every Android app runs in its own process, which is a fork from Zygote. This process contains its own VM, in which the app is run. Android does not use the standard JVM, but uses ART.

2.3.2. Environment

In order for a Java program to run, it needs an environment to store data. This environment is called the Java Native Interface Environment (JNIEnv). The JNIEnv is a pointer to thread-local data, and it contains a pointer to the function table which contains interface functions. This is graphically shown in Figure 2.2. Where the interface pointers on the left, all point to the same table of JNI functions on the right. The JNIEnv is thread specific, and this means that an app can contain multiple JNIEnvs. Multiple JNIEnvs are only used when multi-threading. For every JNIEnv it is possible to obtain the JavaVM. A call to the table of JNI functions is made, to call `GetJavaVM`. All JNI functions need the JNIEnv in order to function. Another type of functions, are the functions of the invocation API [2]. The invocation API deals with loading the JavaVM into applications, and the JNIEnv might not yet exist. With the invocation API the JNIEnv can be obtained from the JavaVM.

2.3.3. Types and data structures

The arguments and return types of Java have to be mapped in order to be usable in native code. In Java there are two different types, namely primitive and reference types. This type difference is kept during the mapping and will be explained separately. Java types in native code are called native types.

Primitive types

The mapping of primitive to native types is shown in Table 2.1. For every Java type there exists a native type which has the same size as its Java counterpart. This is needed as a non-JNI native type with the same name as a Java type can differ in size. Table Table 2.1 lists the mapping for all primitives between Java and native types [38].

Reference types

All types that do not fit into the primitive type category are a reference type. Inside the reference type a few distinctions are made and subtypes are specified. The mapping for these subtypes is shown in

[2] `https://docs.oracle.com/javase/9/docs/specs/jni/invocation.html`

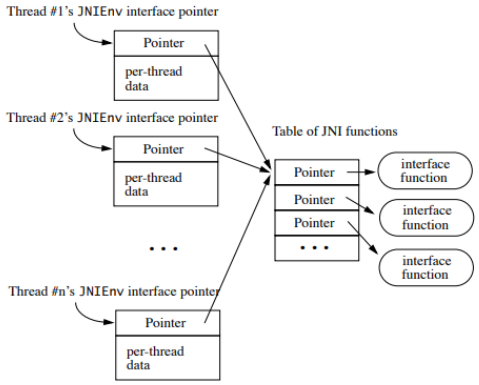


Figure 2.2: How the environment and threads work together in the JNI [3]

Java type	Native type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits floating point (IEEE 754)
double	jdouble	64 bits floating point (IEEE 754)
void	void	Non JNI specific void

Table 2.1: Mapping of primitive to native types

Figure 2.3. Every type that does not fall into a subcategory is of type jobject. A Jobject is the superclass for the other reference types. Primitive types do not inherit jobject, but their array form does.

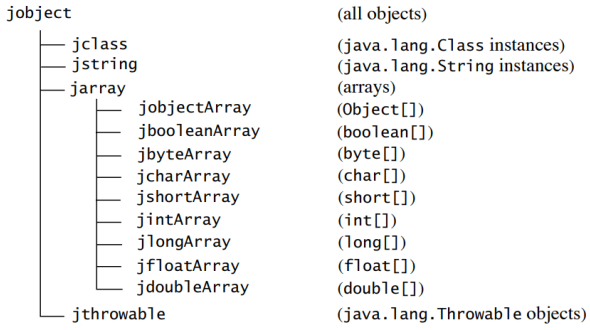


Figure 2.3: Mapping of reference types from Java to C/C++ [3]

JValue

A JValue is a union of all primitive types and the jobject, a JValue can hold all Java types. The definition for the union is shown in Listing 2.3.

```
typedef union jvalue {
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
```

```
        jdouble d;
        jobject l;
    }
```

Listing 2.3: The JValue union

2.3.4. Native methods and native functions

Both the terms native method and native function are used in this thesis. Although they seem similar, they both have a different meaning in the context of the JNI. A native method is a Java method with the native keyword, meaning there is a function in C/C++ that holds the body of that method. The native function is the native counterpart of that Java method. In Listing 2.4 the native method addition can be seen with its accompanying native function in Listing 2.5. The naming convention for linking both functions is used, which is explained by Oracle [39].

```
1 package foo
2
3 public class bar{
4     public int native addition(int a, int b);
5 }
```

Listing 2.4: Native method in Java

```
1 jint Java_foo_bar_addition(JNIEnv *env, jobject self, jint a, jint b)
2 {
3     return a + b;
4 }
```

Listing 2.5: Native function in C

2.3.5. Jobject self

When a Java method calls a native function, the second argument is always a jobject called self. According to the method type, this object can hold different information.

- For an **instance method** it holds a reference to the object from which the method was invoked. This is similar to *this* in C++/Java [3].
- For a **static method** it holds a reference to the class in which the method is defined. This means a static method will require a jclass as the second argument.

2.3.6. Calling a method

The JNI uses an identifier for methods called a MethodID. The MethodID itself is a pointer to a struct called ArtMethod. This struct contains metadata for the method, for example, the declaring class and the item offset in the dex file [40]. The contents of this struct differ slightly per Android version [3]. Only some members of the ArtMethod struct will be referenced in the coming chapters, and only those will be explained below.

access_flags_

The access flags denote which method modifiers apply to this method. In Java, method modifiers can be used for various purposes. The most used method modifiers by Java [41] are shown in Table 2.2. All method modifiers for Android 6 are shown by the Android Open Source Project [42].

entry_point_from_jni_

The entry point from JNI is accessed from the execution stage of the compiler. It gives an entry point to the memory address of a native method. It holds a pointer either to the native function itself, or to a method that can resolve the function in question. The native function can be executed as the memory address of the function is obtained. When the method is not native, this value is 0.

[3]The ArtMethod class for Android 6: https://android.googlesource.com/platform/art/+/refs/tags/android-6.0.1_r81/runtime/art_method.h#541

Modifier description	**Keyword**
Access modifier	*private, protected* and *public*
Restricting to one instance	*static*
Prohibiting modification	*final*
Modifier to require overriding	*abstract*
Modifier to prevent entering from multiple threads	*synchronized*
Modifier for native method	*native*

Table 2.2: Method modifiers used by Java

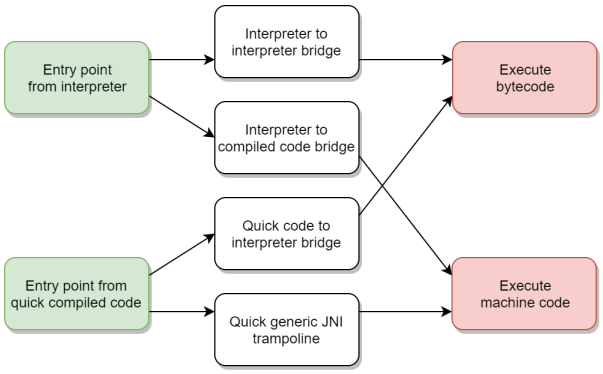


Figure 2.4: A simplified view of the flow for the different entry points. Whereas an entry point is green and the execution of a method is red.

entry_point_from_interpreter_
The interpreter is the starting point for methods that are initiated from a method executed by the interpreter. The current method might not need the interpreten. In Figure 2.4 both paths for executed with and without the interpreter are shown.

When a method needs an interpreter, an interpreter to interpreter bridge is used. This bridge leads to executing bytecode.

When the method can be executed via machine code, a bridge to the quick code compiler is used.

entry_point_from_quick_compiled_code_
The quick code is the starting point for methods that are initiated from machine code. There are two paths from this entry point, as not all methods have machine code. These path can be seen in Figure 2.4.

When the current method can also be executed as machine code, the quick generic JNI trampoline is used. This trampoline is used to obtain the memory address of the method, after which the method can be executed. To obtain the address of a native function, the entry_point_from_jni_ is used besides the trampoline.

When a method has no machine code, it needs to be executed by the interpreter. For this a bridge to the interpreter is used.

2.4. Android Debug Bridge

In order to communicate with an Android device from a computer, the command-line tool Android Debug Bridge (ADB) is used. ADB provides multiple functions, from debugging and installing apps to accessing the shell. ADB is a client-server program provided by the makers of Android [43] that contains three components.

- The **client** runs on the computer and sends commands to the server. These commands always start with the *adb* keyword.
- The **daemon (adbd)** runs on the Android device and executes the command send from the client. It runs in the background of the device. The daemon has to be enabled by the user and every new computer that wants to connect has to be accepted by the daemon.

- The **server** runs in the background of the computer whenever adb is started. It enables the server and the client to communicate with each other. The server can be connected with multiple devices and daemons. In the command from the client the target device is specified with the command. The server will then relay the command to the correct daemon.

In Appendix A the main commands used during this thesis are given.

2.5. Kotlin

In 2017 Kotlin was added as an official language for Android app development [44]. Kotlin is a language build on the JVM and compiles into Java. It is possible to have certain classes in Java and others in Kotlin inside the same Android app. When calling a native function from Kotlin instead of Java, there is no difference from the native side. This means that when mentioning the JNI together with Java, this would also work for Kotlin. However, Kotlin will not be specifically mentioned in the upcoming chapters.

3

Android reverse engineering

The work in this thesis should be put into context with the current work on Android behavioural analysis. This chapter presents the current research and gives an introduction to Android reverse engineering and its use cases. The chapter starts with setting a boundary by defining reverse engineering. Afterwards, different techniques for doing reverse engineering are discussed. Lastly, use cases for reverse engineering with behavioural analysis are given.

3.1. Static and dynamic analysis

Reverse engineering aims to restore a higher-level representation of software in order to analyse its structure and behaviour [45]. Multiple reverse engineering techniques exists, each with its own strengths and weaknesses. These techniques can be split into two main categories namely, dynamic and static analysis.

3.1.1. Static analysis

With static analysis the structure of a program is analysed without executing it [46]. Static analysis focusses on analysing the structure rather than the behaviour of a program. All possible control paths are explored during the analysis, and the structure can be mapped. Paths that are hard or impossible to reach at runtime are still included. It is difficult to deduce how frequent a control path is executed through static analyses. Static analysis for Android is mostly used to obtain an approximation of the source code.

3.1.2. Dynamic analysis

In contrast to static analysis, with dynamic analysis the properties of a running program are analysed [47]. Dynamic analysis focusses on analysing behaviour rather than structure. Only the control paths that were executed at runtime are analysed and not all control paths have to be explored. This means that multiple executions can give different results. Due to the runtime analysis, path traversal frequencies are observed.

Doing static analysis to obtain the structure before dynamic analysis can be beneficial for the results. However, depending on the use case good results can be obtained without having (an approximation of) the source code.

3.1.3. Conclusion

The strengths of static analysis are in recovering structure where the strengths of dynamic analysis are in behavioural analysis. This thesis focusses on behavioural analysis, dynamic analysis seems the better fit. In the coming sections relevant literature will be discussed regardless the type of analysis used. From this a final conclusion can be drawn, based on the relevant literature.

3.2. Methodology

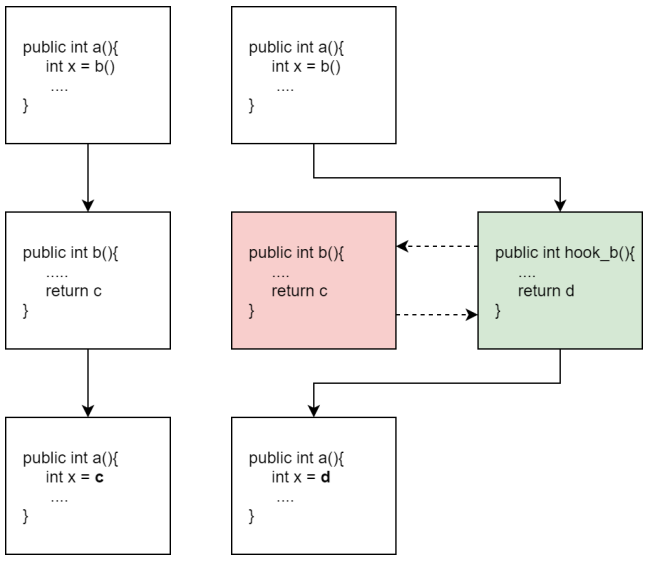
Multiple methodologies exists for doing analysis on Android apps. The following parts contain methods that are frequently used in current research on behaviour analysis. An explanation of these methods, together with some relevant literature on behaviour analysis is given. These techniques and their usage on Android will be covered in the applications section.

3.2.1. Dynamic binary instrumentation

DBI is a type of dynamic analysis on a binary whereby the analysis code is added to the original code of the binary at runtime [48]. A DBI tool will give you an API to hook into the binary and this can be used for the sole purpose of reverse engineering. The tool dynStruct [49] was developed to recover the structure of x86 binaries and to analyse their memory usage. The data needed to recover structure and analyse the memory usage was obtained with DBI. It was used to monitor allocations, access, function calls, context and recording data and output. This data could then be used in an extensive analysis to recover structure and to do an analysis of the memory usage. Brandolini [50] developed an automatic hooking tool that could check if an app was reading device information, detecting user location, used deprecated methods or used http instead of https. It was built on top of the Android Dynamic Binary Instrumentation Toolkit [51].

Method hooking

When DBI places hooks on method level, this is called method hooking. It is a technique where a method is swapped at runtime with a newly created method. The original, swapped out, method can still be invoked, but this is not required. This way the same method call is done by the application, however another function is executed. Behavioural analysis can be conducted with method hooking by calling the original method and placing logs around it. Figure 3.1 shows a graphical representation of method hooking. Where Figure 3.1a denotes the flow of a normal program. Figure 3.1b shows the flow of a program with a hook, where the red and the green blocks are respectively the original and newly created method.



(a) Normal (b) Hooking method b

Figure 3.1: Graphical representation of method hooking

3.2.2. Dynamic taint analysis

Another type of dynamic analysis is dynamic taint analysis which consists of marking and tracking certain data in a program at runtime [52]. The data that is marked is called a tainted source and they are mostly user or sensitive data [53]. It can be observed which parts of the app are affected by those tainted sources. Possible use cases are vulnerability checking due to incorrect user input, or protecting

(user) data. Static taint analysis is also done, where the code is statically analysed in order to obtain which part of the code are tainted. The usage of taint analysis for behavioural analysis will be discussed with its applications.

3.2.3. Fuzzing

With fuzzing new inputs are randomly generated for an application [54]. Fuzzing can be done with and without source code which is called respectively white box and black box fuzzing. Both modify well-formed inputs and test the application with it [55].

Research on fuzzing an Android app has been done. White box fuzzing has been used for security testing of Android apps [56]. A limitation of their work is, it needs manual input to manoeuvre through the app. It has been shown that fuzzing is possible with automated interactions in the user interface [57]. During execution the system calls are captured and analysed. This results in only external and no internal behaviour being analysed. Fuzz testing has been used in order to execute black box testing on Android [58]. Test cases are generated and send to the application, the application is then monitored for bugs. This method is not suitable as it does not focus on the behaviour of an Android app. Black box fuzzing with specially crafted media packages can be used on Android media players [59].

To conclude some research has been done on fuzzing Android apps. Currently, Android fuzzing is used for observing changes or a crash in the user interface. User interface changes are not linked to internal behaviour. Linking behavioural analysis with fuzzing has not yet been researched, and it is not known if it is feasible. In order to extract internal behaviour other techniques have to be used.

3.3. Use cases

With behaviour analysis multiple types of behaviour can be observed from Android apps. Currently, behavioural analysis research focuses on malware, dead code and countermeasures. The state-of-the-art on these use cases will be explained.

3.3.1. Malware analysis

An application of behavioural analysis on Android devices is detecting malware. Multiple studies on detecting and analysing malware were conducted. Droidscope [60] is a DBI tool developed for analysing Android malware. An emulator was constructed to extract information from the Operating System (OS). The OS instrumentation extracts system calls, processes, threads and a memory map. With the information Dalvik instrumentations, Java objects and the Dalvik state could be reconstructed. Droidscope is based on Dalvik, the old runtime environment for Android, which has been replaced by ART in 2014.

ANDRUBIS is another Android dynamic analysis framework for Dalvik [61, 62]. The core of the analysis is dynamic, this is complemented with static analysis. The static analysis consists of analysing the Android manifest, parsing meta information, examining bytecode and identifying permissions. Dynamic analysis is both executed on the OS and Dalvik level for a complete behavioural overview. For dynamic analysis multiple techniques like taint analysis, method tracing and emulator inspection are used. TaintDroid [63] is used for taint analysis, and monitors privacy sensitive information. Multiple sources are tracked simultaneously, for a system wide flow tracking tool. Method tracing works by tracking the invoked Java methods, their parameters and return values and this is combined with the static analysis. Emulator introspection is conducted where the system is monitored from the outside and system calls are tracked. Stimulation techniques to exhaustively explore the app are used, beside static and dynamic analysis.

Another Dalvik based approach by Spreitzenbarth et al. [64] uses both static and dynamic techniques. The static analysis consists off virus total [1] hash matching, analysing the Android manifest and smali code, filtering intents, services and receivers. The dynamic analysis is based on Droidbox [65] which uses both taint analysis and API hooks. Besides the usage of Droidbox it also does native code tracking and network traffic logging. Droidbox [65] is an application that extends TaintDroid [63] with logging all accessed data to the system log. DynaLog [66] is a framework that does dynamic analysis in order to characterize Android applications. With these characterized applications malware can be detected. The framework extended DroidBox to log events and actions, and to monitor,extract and log API calls.

[1] https://www.virustotal.com/

A limitation of the tools discussed is that they only support devices that run Dalvik. ART has replaced Dalvik in 2014, and works on different principles. Non-trivial changes have to be applied to work on ART. The difference between ART and Dalvik is explained in Section 2.1.3. It is not possible to use the tools on current devices, without major adaptations. However, their methodologies can still be used.

Another type of dynamic analysis is done by Crowdroid [67] which monitors Linux kernel calls. The system calls are then parsed and a system call vector per user interaction is created. On these vectors clustering is done with a partional clustering algorithm. The paper by Wagener et al. [68] uses system calls as well to analyse malware, however this is done on 32 bit Windows. Analysing system calls for detecting malware works as there is a difference between system calls in the legitimate and malicious apps. The system calls are mostly compared between a malicious application and its benign counterpart. This is only feasible when both versions of the app are available. In the paper by Wang et al. [69] they analysed the behaviour of Android apps according to static features. These features are component names, requested permissions, hardware and software requirements, filtered intents, restricted API calls, used permissions, certificate information, strings, payload information, code patterns, and suspicious API calls. The feature sets where used in four different classifiers, namely Logistic Regression, linear SVM, Decision Tree, and Random Forest. These four classification methods are compared to determine the best classifier, logistic regression gave the best results. Most features are app specific for example requested permissions, hardware and software requirements and certificate information. These cannot be used to distinguish between parts of the same app. The features that are not app specific could all be obtained in a dynamic fashion, for example requested API calls and used permissions.

Using only static analysis for malware detection has been researched. Arp et al. [70] build DREBIN, where features are extracted from metadata and bytecode. These features are stores in a vector space and machine learning on these vectors is done. Mariconti et al. [71] extracts API calls, the sequence of calls is analysed to model the behaviour. Malware detecting is based on the assumption that the calls in malware differ from a benign app calls itself and their ordering. DroidAPIMiner by Aafer et al. [72] shows another approach by analysing API calls for malware detection. The analysis is based upon extracting features from bytecode that are used frequently by malware. Both in the papers about MAMADROID and DREBIN the authors note that using static analysis alone is not enough and dynamic analysis support has to be added. In the paper by MAMADROID a comparison with DroidAPIMiner has been done and it was mentioned that MAMADROID outperforms DroidAPIMiner.

3.3.2. Countermeasures

Another application of behavioural analysis is detecting countermeasures. Countermeasures are security mechanisms Android apps use to protect itself against treats. Host Card Emulation (HCE) Android apps from the play store were analysed and the countermeasures are divided into categories [73].

- **Anti-analysis** aims to make it harder for an attacker to analyse the application. This is mostly done by obfuscation and Android packing.
- **Anti-rooting** aims to detect when a device has been rooted. When a device is rooted the user has the highest privileges possible on the device. This power can be misused if malware is installed.
- **Anti-instrumentation** aims to detect instrumentation techniques. When an attacker can successfully instrument, other countermeasure like root detecting can be circumvented. In essence an attacker has full control over the application when instrumentation is possible.
- **Anti-tampering** aims to protect the code and data of the application from being tampered with by the attacker. This is done by detecting when code or data has been changed and stopping the execution of the app.
- **Anti-cloning** aims to protect against cloning of the app to another device. When it is detected that the app is running on another device the execution should stop.
- **Anti-key-recovery** aims to protect the cryptographic keys of the application. This is done by hiding them from plain sight, mostly by using white box cryptography.

The behaviour of a countermeasure differs per category and a distinction is needed. Countermeasures for native Intel/AMD x86 binaries were researched [74]. The categories are roughly the same as before,

and the discrepancy can be explained by the difference between Java and native code. The security of four HCE payment apps were assessed with a combination of static and dynamic analysis Kaur et al. [75] . The static analysis consists of extracting smali to get the Java methods and classes in order to obtain the logic of the application. YARA rules are used to to identify packers, protectors, obfuscators, and anti-VM checks. The automated pentesting framework MobSF [76] was used for both static and dynamic analysis. More dynamic analysis was done in order to find security checks and to patch an repackage the app. Afterwards, more analysis is done with intercepting network traffic between server and app, step by step runtime debugging, analysing backup files and memory contents.

It can be observed that papers concerning countermeasures mostly target one specific countermeasure. How Android rooting works, how it can be detected and how it can be evaded has been researched [34, 77]. Nguyen-Vu et al. [77] tried to evade root detection mechanisms. They first investigated what root is and how this can be detected, in order to evade this with hooking both Java and native code. With hooking it can bee seen if a method is executed and the behaviour of a methods can be altered at runtime. Sun et al. [34] made an automatic root detection evasion system. The system works on hooking both Java and native code calls. These calls are logged and the output is manipulated to bypass the root check. The code from both frameworks was not published. The usage of packers [78–81], obfuscation [82, 83] and white box cryptography [83] in Android apps and malware has been studied as well.

3.3.3. Dead code detection

Another application is dead or non-functional code detecting. A dead code removal tool was developed by Boomsma [84]. The tool is made with PHP and the concepts used are specific for dynamic languages on the web. The data was extracted using dynamic analysis, and looks at which files are used and unused. The unused files are then classified as dead code. Chen et al. [85] designed a C++ data model that enables dead code detection. A static analysis on reachability was done in order to detect dead code. The analysis start at the different entry points of the program and finds the closure set of entities reachable. By taking the difference between the whole set and the closure set of entities the unused program entities are obtained. These unused program entities are the parts of the program that are dead code. These methods are not applicable to Android as they use language specific features. The existing research does not use dynamic analysis, but static analysis. There is the possibility to use reflection, as the code is written in Java. Barros et al. [86] defines reflection as a metaprogramming mechanism that enhances the flexibility and expressiveness of a programming language. Its primary purpose is to enable a program to dynamically exhibit behaviour that is not expressed by static dependencies in the source code. Landman et al. [87] states real Java code frequently challenges limitations of the existing static analysis tools. Using static analysis for dead code detection could mean that not all function calls can be observed. However, using dynamic analysis to see which methods are not executed is similar to the method by Boomsma [84]. Dead code is also used inside malware for obfuscation purposes [88].

3.4. Conclusion

This chapter discussed the two main types of reverse engineering, namely static and dynamic analysis. It was found that dynamic analysis was more applicable for this thesis due to its focus on runtime. Multiple methods were discussed together with the current research. In Table 3.1 a summary of the results for the methods is shown. With the obtained information, an answer to RQ1a: ”What current research is available that analyses the behaviour of an Android app?” can be given. Dynamic Binary Instrumentation (DBI) is the only technique that is sufficient to conduct behavioural analysis from a complete Android app. Besides being used for behaviour analysis on Android, it is scalable and generic.

	Used for behaviour analysis	Scalable	Generic	Used on Android
Dynamic Binary Instrumentation (DBI)	Yes	Yes	Yes	Yes
Dynamic taint analysis	Yes	No	No	Yes
Fuzzing	Unknown	Yes	Yes	No
Static analysis	No	Yes	Yes	Yes

Table 3.1: Summary of the discussed methods

4

Current tooling

In Chapter 3 the different methodologies for behavioural analysis were discussed. The current state-of-the-art research together with their tooling was discussed and it was concluded that they would not suffice. Most of the tools were not applicable for behavioural analysis or were made for Dalvik. Changing those tools to work with ART would not be trivial and would require rewriting most of the tool. More on the difference between Dalvik and ART can be found in Section 2.1.3. Other applicable tooling did not publish the tool and/or source code. Another tool to conduct the research has to be found. This chapter discusses and evaluates the currently available tooling.

4.1. Frida

Frida [89] is currently the most used tool for Dynamic Instrumentation on Android. It supports instrumentation for both Java and native code and is based upon method hooking. Frida aims to be easy to use and hooks are written by the user in Javascript. Javascript adds extra complexity as JavaScript cannot communicate with the JNI or Java directly. Frida uses a so called engine, to use JNI functions to communicate with Java. However, the hooking functionality of Frida is easy and relatively stable. It also has good support for current Android versions and is actively maintained. Frida does need root acces, but there are guides to circumvent this by packing the required binary inside the app [90].

4.2. Xposed

Xposed is a framework that allows the user to install modules that can change both Android apps and system level functionality. Xposed works by replacing Zygote with a modified version [91]. Due to the fact that Zygote needs replacement, it is flashed onto the device. This flashing is done via custom recovery and needs besides root access also an unlocked bootloader. Xposed is not maintained beyond Android 8, however there do exists forks that support Android 9 and 10. Due to all these modifications, Google SafetyNet which is part of Google Play services can detect Xposed. Google SafetyNet is an API by Google, which allows an application to assess the security and health of an Android device. The testing is handled by servers from Google [92]. Xposed does changes on a operating system level, it can break devices from providers (like some Samsung) that change Android quite extensively [93]. There are some unofficial forks that solve this issue, but this can be a problem for less used device providers. With Xposed it is possible to hook methods in Java and native code.

To conclude Xposed has the benefit of not only being able to hook Java functions, but also native. The downside is that it needs to be flashed on via custom recovery, and might not work with Android phones that were customized by their providers. Furthermore, it is detected by Google Safetynet and is currently not maintained.

4.3. Cydia Substrate

Cydia Substrate [94] is a Dynamic Binary Instrumentation tool that supports both Android Java and native method hooks. It supports only Android versions 2.3 up to 4.3. This means it is limited to the

Dalvik runtime and cannot be used with current Systems. Cydia is unable to meet the requirements for this research.

4.4. Dynamorio

Dynamorio [95] is a DBI tool that supports Android. It focusses on doing DBI on multiple operating systems and architectures. It is made for binary instrumentation on multiple instruction sets and focuses on native code. Some work has been done to support Java apps, however this is not in an advanced state. Dynamorio is not suitable as Android apps are mostly made out of Java and not always contain native code.

4.5. ARTist

ARTist is an Android Runtime instrumention tool [96]. ARTist is a tool that supports ART and has a different approach then most. Instead of changing the app via another binary, the compiled app is changed by recompiling with its own compiler. ARTist hooks on instruction level opposed to method level. At the time of writing, ARTist is still in beta and is not stable. Furthermore, it currently only supports Android 7 and 7.1 [97]. ARTist changes code by using its own compiler. To extend or improve ARTist knowledge about compiler construction is needed. Understanding how the Android compiler works and how the compiler of ARTist can be adapted when an error arises, is very complex. In the paper [98] two use cases are shown. Firstly, it is used for dynamic permission enforcement. Secondly, it is shown how intra-app taint tracking can be done.

4.6. ARTDroid

ARTDroid is an Android hooking framework based on ART [99]. The source code for this tool is available at Github [1]. However, the tool has not been updated since 2017 and has only good support for Android 5. In the code are multiple references to not working functions or functionality that still has to be implemented. The documentation pages were taken down and setup instructions are lost. The project is in such a preliminary stage, that understanding the code and setting up will take more time than constructing a new framework. ARTDroid is not suitable for this project, as there are no instructions and the project is not finished. A case study was shown in the paper [99], where one function was hooked.

4.7. Other tools

On Github there are a lot of projects that aim to hook Android apps. The problem with these frameworks is, that most have only been made for a specific device or Android version and are difficult to upgrade. Even more so, if there is documentation it is mostly written in Chinese which adds an extra challenge to these projects. Quite a few of these tools were tried, but as they were not sufficient they have not been added in this section. It could be that there is a tool that would work, however it is impossible to try all tools available on Github. The tools that were discussed above are tools that have been used a lot or that are from recent papers.

4.8. Tool selection

From all currently available tools Frida is the only stable maintained tool that provides Java hooks for Android. Extending a tool is a possibility. Extending the Dalvik tools is not a trivial task and can be more work than making a framework from scratch. Very little information can be re-used and first the internals of the tool have to be studied and understood. Xposed already supports new Android versions, and the conversion from Dalvik to ART is not needed. However, as Xposed changes Zygote the changes are going much deeper than what is required for the runtime analysis. Extending ARTist means that specific knowledge on compiler construction which also provides an extra challenge. Lastly, extending ARTDroid is difficult due to the lack of documentation and still to implement functionality. From all tools Frida is the most suitable.

[1] `https://github.com/vaioco/ARTDroid`

	ART support	Supports Android 6	Maintained	Java support	Native support	Complexity of extention
Frida	V	V	V	V	V	-
Xposed	V	V	X	V	V	X
Cydia Substrate	X	X	X	V	V	X
Dynamorio	V	V	V	X	V	X
ARTist	V	X	V	V	V	X
ARTDroid	V	X	X	V	V	X

Table 4.1: Different capabilities per tooling

4.9. Proof of concept Frida

In the previous section it was concluded, that Frida is currently the best Android Dynamic Instrumentation tool. With Frida a Proof of Concept (PoC) experiment was conducted in order to evaluate if it satisfies the research questions.

4.9.1. Frida script

In order to conduct the experiment with Frida a script was written. This script provided a means to hook all methods in an Android app. With the script all classes, functions and overloads are traversed to create a hook for every method. A method hook consist of logging statements around the invocation of the original method. There are two logging statements, the invocation and its arguments are logged when the method is invoked. When the method returns, this is logged with its value. This is shown as pseudocode in Algorithm 1, the full script can be found in Appendix B.

With the script the behaviour of an Android app can be analysed as all internal behaviour is logged. The print statements are printed on the terminal, which can be written to a file. The log statements itself were with clear separations, in order to make it transferable to another format with regular expressions.

Algorithm 1 Pseudocode for Frida script

```
1:  classes ← all classes in Android app
2:  for all class ∈ classes do
3:      methods ← All method names from class
4:      for all method ∈ methods do
5:          overloads ← all overloads per method
6:          for all overload ∈ overloads do
7:              Overload function overload
8:              function overload of overload(params of overload)
9:                  print name and arguments of overload
10:                 result ← invoke the real function and save result
11:                 print result
12:                 return result
13:             end function
14:         end for
15:     end for
16: end for
```

4.9.2. Execution

When using the Frida script, only a limited amount of hooks could be placed. An experiment was conducted (described in Section 6.4.2) and showed that Frida can only hook around 5000 methods before the Android systems deems the app unresponsive. The small app, Rootbeer Sample [2], that checks if a device is rooted contains over 60000 different methods. This means that Frida cannot hook all methods of Rootbeer at the same time. We can create multiple Frida script that hook a different subset of functions every time. This would result in at least 13 different scripts of 5000 methods, to

[2] https://play.google.com/store/apps/details?id=com.scottyab.rootbeer.sample

hook more than 60000 methods. It would be impossible to build a call-graph of all methods, in order to see how a method was invoked. If this was to be done with Frida, a lot of static analysis has to be done to find overlapping subsets. The subsets need to be overlapping as in the end all function are executed in a chain. Which would result in a lot more than 13 executions, due to the increase in subsets.

A lot of pre- and post-processing is needed to extract the data that is necessary to detect the different types of behaviour. Conducting experiments on real world applications is almost impossible as the amount of functions is a lot higher. For a small app most functions come from Android libraries and are a baseline. For comparison, Rootbeer Sample has around 4500 classes, whereas the Android browser Google Chrome has 7000.

4.9.3. Extending Frida

Frida has some limitations that need to be addressed when using it for large scale behavioural analysis. Frida has documentation on the usage of the framework. However, the documentation for developing the tool is very minimal and the code itself is quite technical. It contains a lot of (hexadecimal) numbers without an explanation which make it difficult to understand. The fact that the code is written in both JavaScript and C++, adds an extra factor of complication. The JavaScript and C++ parts are connected via proxy functions which are not straightforward. For quite some time Android was not a first class citizen, as Frida was initial developed for iOS. Although, a lot of work has been done this left some artefacts. Especially in the C++ part, there is no dedicated Android class and its code is scattered across files for other architectures.

The fact that Frida uses JavaScript gives the tool a lot of overhead. This overhead makes that the tool cannot be used for large scale analysis. Removing the JavaScript part of Frida would remove almost all functionality. This architectural decision of using JavaScript makes that the whole framework has to be rewritten in another language in order to remove the overhead.

Frida is designed in a way to be as user friendly as possible, by providing the JavaScript API. This whole process provides so much overhead that it is impossible to hook all functions. The JavaScript to C++ conversion will always introduce overhead, as a conversion needs to be made. Hooking all function in an Android app is impossible, due to the introduced overhead.

4.10. Conclusion

This chapter discussed the currently available tools and it was concluded that Frida was the only tool that would be able to conduct the analysis. Further experiments with Frida were conducted to see if Frida was capable to conduct the analysis.

It was shown Frida has quite a few limitations that make it impossible to hook all methods in an Android app. Furthermore, it is shown that is very difficult to extend Frida and that some design decision prevent certain options. It can be concluded that Frida is not suited for the tasks of analysing the runtime behaviour of Android apps. It is impossible to hook every function of an Android app simultaneously. This means a new tool has to be developed that removes the overhead that Frida has in order to study the behaviour of Android apps. The benefit of making a new tool opposed to extending an existing tool, is that only the concepts of Android have to be studied.

5

Samaritan

In order to conduct behavioural analysis a new tool was developed. This chapter explains the functionality as well as the underlying structure of the tool. The tool created is called Samaritan [1].

5.1. Functionality

Samaritan is a method hooking (see Section 3.2.1) tool that is able to place hooks in both Java and native code on Android. In this section the functionality of the tool is discussed.

5.1.1. Hooking a function

The core functionality is hooking a function, these hooks can be placed in multiple ways. Firstly, a user can place a hook that is totally customisable. The original method can be called from within the hook. The user has to write hooks for both java and native in C++.
Secondly, a script can be used that hooks all Java methods of an app at once. These hooks are predefined and places logging statements around the invocation of the original function. The invocation with its arguments as well as the return with its value is logged. These logs can be converted to DOT files, with an included python script.
Details about the internals of these functions will be given later on.

5.2. Internals

The hooking framework consists out of four different parts. These parts are connected, but are separated due to their functionality.

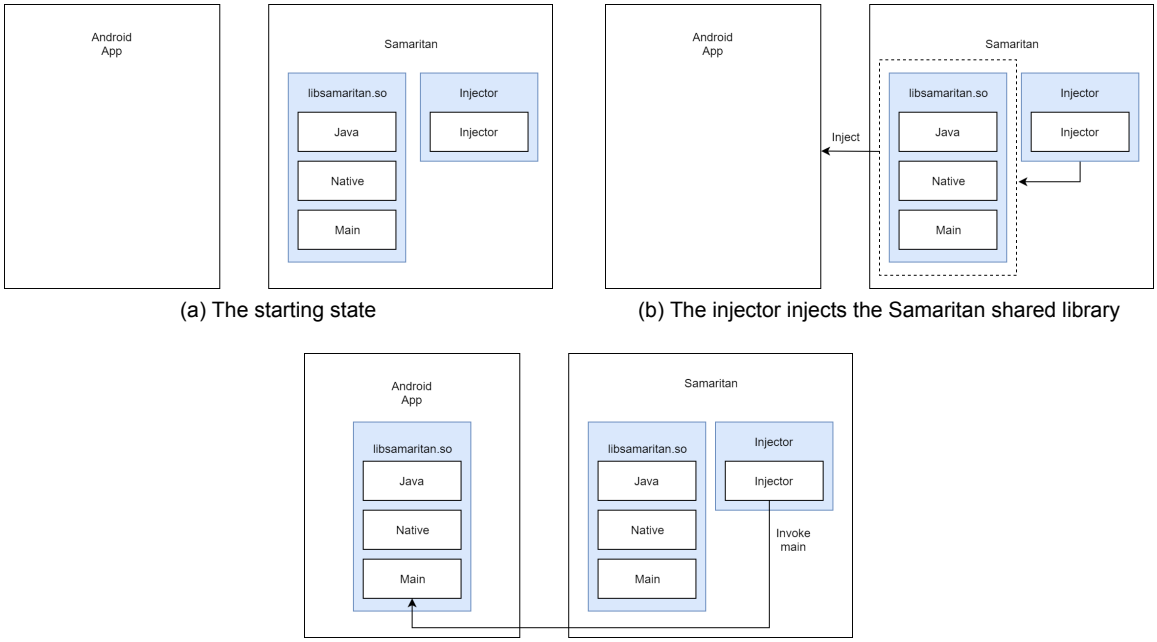
- The **native** part places hooks into native code.
- The **Java** part places hooks into Java code.
- In **Samaritan** itself, referred to as main, the user specifies which hooks need to be placed. Samaritan will then call the appropriate functions in the other parts.
- The **injector** injects the other parts into the Android app and invokes.

5.2.1. Injector

To place a hook, access to the address space of the application is needed. This access can be obtained by injecting a shared library and loading that library inside the app. For this Samaritan is compiled as a shared library, with the native and Java parts included. A general overview of the injection process along with an explanation is shown in Figure 5.1.

The steps needed to inject the app are as follows:

[1] This name is a reference to an all knowing and manipulating artificial super intelligence in the series Person Of Interest. It acted as a mass surveillance system. The newly created tool will do mass surveillance on Android apps, and this name was chosen.



(a) The starting state

(b) The injector injects the Samaritan shared library

(c) The injector invokes the main of Samaritan

Figure 5.1: General process of injecting Samaritan into an Android app

1. **LD_PRELOAD** Before the shared library can be injected, it has to be first loaded in the system. This is done by using LD_PRELOAD. This way the dynamic linker can resolve the symbols and bind them at runtime.

2. **Attaching**
   In order to inject our shared library inside the Android app we need to be able to remote call functions and for this ptrace is used. Ptrace is a linux system call that provides an interface to control and change the memory and registers of another process [100]. In order for ptrace to function SELinux has to be set to permissive. It is also possible to set a label on the device to enable the injector to use ptrace [101, 102]. This means that currently the device has to be rooted. The ptrace option, PTRACE_ATTACH is used to attach to the app.

3. **ASLR**
   Android uses ASLR to randomize the address space as explained in Section 2.2.4. Some remote functions are needed to inject our library, to invoke them ASLR has to be circumvented. The needed functions are:

   - dlopen, to open our shared library.
   - dlsym, to call a function of our injected library
   - calloc, to allocate a string in the app.
   - free, to release the memory of calloc.

   It has been shown that ASLR can be circumvented and it is based upon two premises [103, 104].

   First, a function has a predetermined offset inside a library. This means that regardless of where the library is loaded the function will always be at base address + X. Second, a library is loaded inside the address space of an app. When a library is loaded by multiple apps, the library is loaded separately for every app. A library has a local address for where the function resides in the app.

   To use these premises, the following values are needed.

   (a) The base address of the library where the function resides locally.

(b) The base address of the library where the function resides in the target app.

(c) The local address of the function.

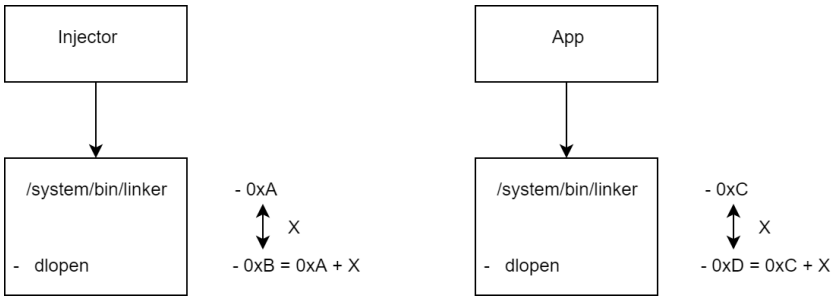


Figure 5.2: The address space of both the injector and the app to inject to. A method always has the same offset X from the base address.

With these values the location of the function in the target app can be obtained. The base address of the local library is subtracted from the local address to obtain the offset of the function. This offset can than be added to the address of the remote library. The result is the address of the remote function.

4. **Opening library**
The remote functions can be called, after bypassing ASLR. The injector calls remote functions to open our shared library inside the app.

The following steps are executed:

(a) Use calloc, to copy the name of the library to the target process.

(b) Use dlopen, to open the library with the address of calloc.

(c) Use free, to free the library name allocated by calloc.

5. **Execute main function**
The main function of Samaritan needs to be called, inside the injected shared library. In the general figure this can be seen in Figure 5.1c. In more detail the remote functions that need to be used are as follows:

(a) Use calloc, to copy the function name to the target process.

(b) Use dlsym, to obtain the symbol for the function to call.

(c) Use free, to free the memory used by calloc.

(d) Use calloc, to copy the arguments of the main function.

(e) Call the symbol obtained by dlsym, with its arguments.

From this point, the main function of Samaritan has been run and all hooks were placed.

6. **Detach**
Samaritan is now injected and running, and the injector is finished. The injector will detach itself from the app, this is done with the ptrace call PTRACE_DETACH.

Figure 5.3: Transforming function A to call function B [4]

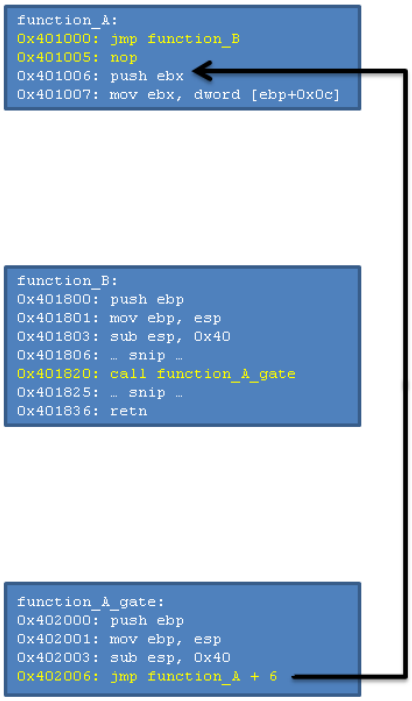


Figure 5.4: Hooking function A, with hook B and trampoline function A gate [4]

5.2.2. Native

The native part is build upon an existing native hooking framework [105]. This framework could hook functions from its own address space. Only small modifications were needed to make it work on the native part of an Android app. Currently, only 32 bit Android devices are supported and this has to be extended to 64 bit.

The Native part method *hook* is used to place a hook. It needs the following values:

- The location of the hooked method
- The address of the newly created hook
- A pointer to a placeholder function with the same signature as the function.

With these values the hooking can be placed. In the following example, the function that is hooked is called *function A* and the new function is called *function B*. Transforming the original *function A* to call *function B* is shown in Figure 5.3.

To call the original method a trampoline function is needed. A trampoline function is a function that replicates the overwritten instructions. The overwritten instructions are the yellow instructions in the original *function A* in Figure 5.3. Afterwards it jumps to the original method at the instruction after the overwritten bytes. The trampoline function will be located at the placeholder. This is shown in Figure 5.4, where *function A gate* is the trampoline [4]. The exact implementation can be found in the original code [105].

To undo a hook, the last few steps are executed in a modified form. The hooked function is restored to its original form.

5.2.3. Java

The Java part has been build from scratch, but the logic was inspired by Frida [89]. The method is based on the fact that MethodId is a pointer to the ArtMethod struct.

The following changes are made:

- **access_flags_** In order to hook a Java method, it is converted to a native method and the access flags need to be adapted accordingly. This is done by adding the access flag for native code and

for fast native code. The fast native access flag speeds up the JNI transitions and disables garbage collection until the method was executed. Methods run with the fast native access flag cannot be suspended. The VM will not suspend during execution of the native method.

- **entry_point_from_interpreter_** The hook is created in native code, therefore the interpreter will not be used. This entry point should redirect to the quick compiled code via the interpreter to compiler bridge. The value of this is obtained from the classlinker in the VM.
- **entry_point_from_jni_** This value is changed to use a pointer to the newly created replacement function. With this the newly created native function can be found and invoked.
- **entry_point_from_quick_compiled_code_** The hook is created in native code and machine code will be used to execute this method. The quick code trampoline is used for machine code. The value of this is obtained from the classlinker in the VM.

5.2.4. Main

The main part of Samaritan is the part of the tool the user interacts with, it invokes both the Java and native parts. The main method of Samaritan is started by the injector. When it is started it will create the hooks for the methods marked by the user. Pseudocode for hook generation is shown in . A closer look to the different ways of generating a hook will be given.

Algorithm 2 Pseudocode for Samaritan

1: **function** hookedMethod(JNIEnv *env, Jobject obj, X1, X2, ..., Xn)
2: $className, methodName \leftarrow$ from previous context
3: log(Invoked: $className\ methodName$ X1,X2,...,Xn)
4: $result \leftarrow$ Call to original method
5: log (result of $className\ methodName\ result$)
6: **return** $result$
7: **end function**

Native

Hooking a function in native code requires the following information. Firstly, the base address of the native library which can be obtained via the native part. Secondly, a newly created hook method. Lastly, the address of the function to hook needs to be obtained. This can be done in various ways for example with the linux function readelf [2] or a decompiler like snowman [3] or IDA [4].

A snippet for placing a hook for the function `check` is showed in Listing 5.1. In this `new_hook_check` refers to the newly created hook. The function address is stored in `check_offset` and `old_check` holds the function definition for the trampoline. All code for creating the hook including the hook is located in Listing C.1.

```
// Method for creating a hook for check
void hook_check()
{
    const char *name = "libnative-lib.so";
    //Function in native part to obtain the base address of the library
    uint32_t base_address = get_addr_range(name).base;
    uint32_t offset = base_address + check_offset;
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "check_offset address %u\n",
        offset);
    //Function in native part, to hook the function
    hook(offset, (uint32_t)new_hook_check, (uint32_t **)&old_check);
}
```

Listing 5.1: Code to place a native hook

[2] https://linux.die.net/man/1/readelf
[3] https://derevenets.com/
[4] https://www.hex-rays.com/products/decompiler/

Single Java hook

In order to hook a Java function a call to the attach function is required, which attaches the hook. A new function has to be declared that will function as the hook. To call the original method two other calls are needed, which apply the patches specified before. In the coming snippet the patches are listed as `before_original_call` and `after_original_call`. This is done for clarity to omit the arguments needed for the normal patching. In Listing 5.2 an example of a hook for the method getButtonText is shown.

```
//Variable holding the original address
uint32_t pointer_getButtonText = 0;

//The hook
jstring j_new_getButtonText(JNIEnv *env, jobject self, jint number)
{

    //Print the invocation
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "Invoked com.example.hellojni.
        HelloJni.getButtonText (i)Ljava/lang/String; with %d", number);

    //Call to update the ArtMethod struct for calling original method
    before_original_call(pointer_getButtonText);

    // Call original method
    jstring str = (jstring)env->CallObjectMethod(self, (jmethodID)pointer_getButtonText,
        number);

    //Call to restore the ArtMethod struct to call hook at new invocation
    after_original_call(pointer_getButtonText);

    //Get a UTF readable string from the jstring.
    const char *java_str = (*env)->GetStringUTFChars(env, str, NULL);

    //Print the returning value
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "Return of com.example.hellojni.
        HelloJni.getButtonText (i)Ljava/lang/String; %s", java_str);

    // Release allocated string
    env->ReleaseStringUTFChars(str, java_str);

    //Return the original value
    return str;
}

main()
{
    ...
    //Attach the newly created hook
    pointer_getButtonText = attach("com.example.hellojni.HelloJni", "getButtonText",
        "(i)Ljava/lang/String;", (uint32_t)j_new_getButtonText);
    ...
}
```

Listing 5.2: Code to place a Java hook

All Java methods

If all functions of an Android app need to be hooked, hooking them one by one would be an infeasible task. Therefore it is possible to hook all functions in a semi-automated fashion. More on this can be found in Section 5.3.

5.3. Hooking all methods

In order for the tool to hook all functions, it is needed to dynamically create functions. To create a hook, a method needs to be created with the same argument and return types. If we want to hook all functions, it is impossible to hook all functions by hand. The code has been written in C++, therefore dynamically generating functions at runtime is not trivial. The arguments are only known at runtime and a hook

needs context (function name, signature) to call the original function. In C++ to dynamically create functions, lambdas can be used. However they are not suitable for this cause. The reasons for this will briefly touched upon.

- A pointer to a lambda can only be obtained when there is no lambda capture. This means no context can be passed trough to the lambda. Furthermore, if you let the type be determined at runtime this will also prohibit taking the pointer of a lambda. A pointer is needed as the function address needs to be set as the JNICode.
- If declaring a lambda in a loop, when the pointer and memory address can be obtained all lambdas have the same memory address.

In C++ there are also std::functions, which will not solve the problem as it needs a declared type on compile time. This type cannot be inferred from the context and the placeholder type specifiers will not work.

5.3.1. Assembly

With C++ it is not possible to dynamically generate functions and a different approach had to be found. The new approach uses assembly for the initial hook, which functions as a layer between the function call and the hooking framework. This assembly code can be automatically generated and will redirect to a function inside the hooking framework. During the generation of the assembly, context about that function is available and this is saved inside the function. On call this context can then be passed trough to the hooking framework, together with the arguments. The arguments are packed inside a list, to reduce the amount of functions needed inside the actual framework.

This general flow is visualised in Figure 5.5. Where the dotted line denotes a function call `A` and the straight line function call `B`.

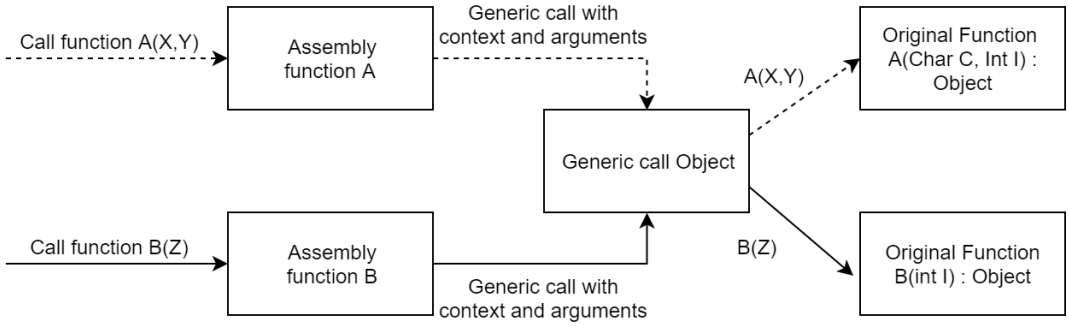


Figure 5.5: Flow of calling a function via assembly

Context and arguments

The assembly code functions in between the call and the actual hooking framework. In order to generate the assembly code, all methods need to be enumerated. During the enumeration context (name of method, name of class and signature) about the method is available. This context can be added to the assembly function, as this information is not available at runtime.

The signature of the method is known during creation, which provides the amount of arguments and their types. Inside the assembly function the arguments are iterated and saved inside a list. This list is stored inside a struct, to enable for changing the deserialization in C++. The list will be handled by C++, this lists can only hold one type. The type of the list is a long long inside C++, as all primitive types can be stored. The long long type has the same size as a long in Java on a 32 bit device, namely 64 bit.

Inside C++ the accompanying context is used to cast the arguments back to its original type. After casting the arguments are saved as a JValue. A list of JValues can be passed as one parameter to the original function.

The intermediate step of using an intermediate type sounds redundant, but is needed. Inside assembly there is no notion of types, only of size. Only a distinction between words and double words

can be made. In C++ these double words are then interpreted as a long long, as this is the largest data that can be transferred. The JValue is a union and a direct cast cannot be made. In order to put a value inside a union the correct member needs to be specified and this cannot be done with a direct cast. An example of an assembly function can be found in Appendix D.

Generic function
The assembly code calls a generic function, depending on the return type. The JNI call that needs to be made is dependent on the return type, and a specific generic function is called. There are 10 generic functions one for every primitive type and one for the jobject type.

The generic function is a bridge for calling the original function. Every generic function goes via the same process and the steps are shown below.

1. The class name, method name and signature of the method are logged.
2. The intermediate type is casted back and converted to a JValue. This is done with the help of the method signature, which was passed as part of the context. After conversion the arguments are logged, as the type is needed for logging.
3. The method is prepared for calling the original method. For this the artMethod needs to be prepared as explained before.
4. Call the original method and store the result.
5. The artMethod struct is restored.
6. Log the resulting value
7. Return the value from the original method.

Dynamic
Using assembly to hook all functions, automates the process of hook generation. However, the assembly files have to be pre-generated and requires pre-compilation. Changing one hook, can result in compiling all assembly files again, which depending on the amount of files could take some time. A solution for this has already been found, although its not yet implemented. Opcodes can be used, as they can be generated and executed on the fly. Opcodes is an abbreviation for operation code, which are codes that specify which operation has to be performed. Assembly can be disassembled to opcodes. This opcode generation can both be done by hand or a framework like keystone [5]. The mechanism works the same as with the assembly files, only the files do not need to be generated and compiled beforehand.

A second problem that needs to be solved for dynamic hooking is looping trough all methods. For this the classloader can be used to obtain all the loaded classes. This step has not yet been completed, but Frida [89] has shown this is possible. Due to implementation differences, it is not enough to simply copy their solution. However, it should be possible to obtain all currently running methods. The JNI is used instead of plain Java and concepts like reflection work differently. Inside Java it is possible to obtain all classes with reflection, and it should be investigated if this can be done with the JNI.

Given that the loaded classes can be obtained and opcodes are implemented dynamic hooks can be made. Due to Samaritan already being able to hook all methods inside of a Java class. These methods can then be hooked via the opcode method described above.

5.4. Output

In order to analyse the behaviour output needs to be captured. The logs are written with logcat and a command can be used to capture all these logs and write it to file. When placing the logs a unique identifier is used, which makes filtering possible. A script has been created that can parse and transform the data to DOT files. This can be transformed to other formats by using regex.

[5] http://www.keystone-engine.org/

6 Evaluation

In the previous chapter, it is shown how a tool for dynamic analysis of Android apps was build. The tool itself has to be evaluated in order to see if it outperforms the current state-of-the-art. This evaluation will be done according to several usecases. The chapter starts with an explanation of the apps that will be used for the analysis. Afterwards, the evaluation of Samaritan will be shown.

6.1. Setup

In order to test a hooking framework both Android apps, as well as a test device is needed. The experiments are all conducted on a LG Nexus 5 running Android 6.0.1. The following existing apps are used in the analysis.

Rootbeer

The Rootbeer sample app [1] is a sample app for the Rootbeer library [106]. It is the most used library for checking if a device is rooted and it aims to include all methods for doing root checks.

The sample app contains only a few UI options. The fact if a phone is rooted does only change a few UI arguments. A good overview of the calls on a typical run can be obtained and the results will be reproducible.

InsecureBankv2

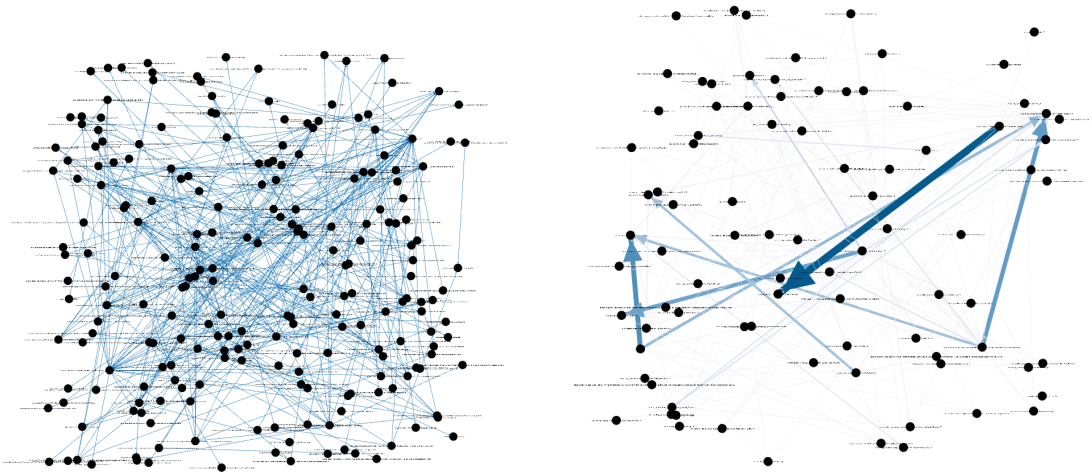
InsecureBankv2 is an app that mimics a banking app, and has some security flaws [107]. The included documentation lists the different security flaws. These security flaws relate to all stages of the process and not all are related to the internal behaviour. There are vulnerabilities in the python backend, which cannot be detected from the app. Vulnerabilities related to exploring all paths cannot be detected too.

6.2. Call graphs

A call graph is a a graph that represents the calls of a program as a graph. The dynamic relationships between the different methods are represented in a static way [108]. Call graphs can be constructed with the usage of static or dynamic analysis and both result in a different graph. With static analysis every possible relationship will be showcased, even when this case will not occur at runtime. Opposed to dynamic analysis where only the executed calls are shown, but this is limited to the calls that occur in that specific run of the program. In order to obtain a full call graph for an Android app, all functionality and UI options should be used in order to get a correct result. Currently, there is no tooling available to create dynamic call graphs for Android. A call graph generated with Samaritan will be compared with a static call graph.

Rootbeer is used for conducting the callgraph experiments due to its limited size and reproducible results. The static analysis is done with androguard [109], which is among others used by DREBIN [70] and MobSF [76]. Only calls that were made from or to a method within the Rootbeer app itself were included for readability. All methods obtained with the static analysis, were used as an input for the dynamic analysis.

[1] `https://play.google.com/store/apps/details?id=com.scottyab.rootbeer.sample`



(a) Static call graph (b) Dynamic call graph

Figure 6.1: The static and dynamic call graphs for Rootbeer

The static and dynamic call graphs can be seen in Figure 6.1. It can be observed that the dynamic graph is more condensed and contains significantly less nodes. The static graphs contains 255 nodes, whilst the dynamic graph only contains 88 nodes. There are also 331 edges opposed to 168. In the dynamic call graph the more calls are made the bigger the size of the arrow and the darker the colour. With the static graph there is no denotion of occurrences possible.

6.2.1. Nodes not in the dynamic call graph

Due to the current limitations of the tool, there were some methods that could not be hooked. Upon a manual inspection 40 from the 71 methods were not invoked. The other 31 were a fault from the tool itself.

From the 73 other nodes missing in the call graph an inspection has been performed why they were not invoked. Around half of the methods were not invoked, due to them executing on start up. Currently, the tool is started manually, and this happens after the init process. There is no core functionality of Rootbeer during this process.

Half of the remaining methods were due to logging and or errors that were not triggered.

An interesting method that was not executed is `com/scottyab/rootbeer/RootBeer isRooted-WithoutBusyBoxCheck`. Manual analysis revealed that this method is not executed at runtime. This method is only used during testing and can be enabled by a user, but is not enabled in the current version of the app. These non-executed methods can be found by comparing all hooked methods with the outcome of the dynamic analysis.

The other methods were Java library functions that were not executed.

6.2.2. Occurrences in dynamic call graph

In the dynamic call graph the frequency of function calls is shown, where the arrow gets thicker the more a function is executed. In table 6.1 all calls that are made more than 100 times are shown.

From this table two edges are related to checking if a device has been rooted.

First, the most occuring edge is a call to check static rooting features [2]. The method loops over properties, to see if the properties `ro.debuggable` or `ro.secure` are changed. These properties can provide an indication that the system has been tampered with.

The other edge related to root checking is the call from `checkForRWPaths` to `equalsIgnore-Case`. This method checks if a path is writeable, which gives an indication if rooting has occurred. Using `equalsIgnoreCase` is a common practise for comparing paths.

[2] `https://github.com/scottyab/rootbeer/blob/593ff5670c7286faf69ca188b977c7d3d1a478c1/rootbeerlib/src/main/java/com/scottyab/rootbeer/RootBeer.java#L253`

From	To	#Calls
com/scottyab/rootbeer/RootBeer checkForDangerousProps	java/lang/String contains	920
com/scottyab/rootbeer/sample/CheckRootTask doInBackground	uk/co/barbuzz/beerprogressview/ BeerProgressView setBeerProgress	874
com/scottyab/rootbeer/sample/MainActivity initView	java/util/ArrayList add	607
com/scottyab/rootbeer/sample/MainActivity showInfoDialog	android/content/res/TypedArray recycle	484
com/scottyab/rootbeer/sample/CheckRootTask doInBackground	java/util/ArrayList add	302
com/scottyab/rootbeer/RootBeer checkForRWPaths	java/lang/String equalsIgnoreCase	289
com/scottyab/rootbeer/sample/MainActivity showInfoDialog	java/lang/String equals	160

Table 6.1: Most frequent edges

6.3. Hard coded analysis

Current root checks are checks that mostly include predefined values or specific calls that are mostly used for this use case. These checks, check for one of the following:

- If a **rooting application** is installed or **files** associated with rooting are present.
- **Build values** are checked. These do not necessarily mean the device is rooted, but it shows the OS is not signed by Google.
- It is checked if certain **commands** can be executed, like the `su` command. This command is used by most rooting applications, to give a user access to root.
- **Permission** and **access** to files and folders is checked. A normal user cannot access some system files, whilst a rooted user might.

Root check detectors have to stay updated with these static values. The root checks currently only change when the rooting apps change, the rooting method changes drastically, Android changes or if a better way of detecting is found. The first one is not a problem as when it is found that an array with packages is expanded that new package can be outputted and added to the program for a new run. There is currently no reason to assume the rooting method will change drastically, as the current methods work fine. When adapting to a new Android version most likely the rooting method will change (slightly). Updates come announced, and after release it should be checked not changes are made. Lastly, new checks can always be added and current analysis of the state of the art has to be done in order to see if the checks adapt. Changes due to any of the above mentioned methods can be observed by manual analysis. For a company like Riscure where they analyse a lot of Android apps, this is something that could be integrated in the workflow.

6.3.1. Analysis

An analysis was done with the most common used words for root detection. These words were obtained from a manual analysis of 35 banking apps.

The results of the analysis can be found in Table 6.2. Occurrences are shown in a non-overlapping manner, so for example *.su* are not included in *su*.

The analysis is compared with an Android app for showing app usage [3]. None of the keywords mentioned above were found during the analysis. Analysing the occurrences of predefined keywords, can give an indication of countermeasures.

[3] https://github.com/googlesamples/android-AppUsageStatistics

Word	Occurrences
ro.debuggable	939
ro.secure	939
ro.build	184
su	6
super	3
selinux	4
cloak	16
root	14
.su	8
hide	8
test-keys	3
which	3
busybox	2

Table 6.2: Words related to rooting and their occurrences

Alternatives

With static analysis these keywords can be found in the code. However, checking if and where a call with a keyword is executed is difficult. This requires knowledge of runtime information. This is made harder by the use of source code obfuscation.

With static analysis it is difficult to see wherein the flow a method is executed. Without this information it is difficult to see if a check is executed at the right time. It could be only executed during a test call, or at an illogical place.

Besides static analysis other dynamic tooling exists. These tools cannot hook all function, which makes it difficult to obtain a general overview. All calls containing or using a certain keyword can be hooked. Hooking only these functions, means that the context of were these functions are executed is lost.

When source code obfuscation is used, knowledge of the original naming is needed to hook the function. The full names are used during runtime, and the obfuscated names cannot be used. Creating a general overview with this information requires removing the obfuscation.

6.4. Comparison between Frida and Samaritan

In this section Samaritan is compared with Frida, the current best tool for Android dynamic instrumentation. The logic of Samaritan is inspired by Frida, however due to language differences the implementation is quite different. This results in different capabilities. Frida and Samaritan will be compared on the amount of overhead produced and the amount of functions that can be hooked.

6.4.1. Overhead

Frida and Samaritan are compared with regards to the introduced overhead when hooking a function. To measure the introduced overhead, an app was created to benchmark the different hooking strategies. The benchmarking is conducted by having a method that calls a hooked method 100.000 times. The runtime of the encapsulating method is printed on screen. The code snippet containing the logic of the app can be found in Listing 6.1. The newly created hook, purely consists of calling the original method and returning its value.

It has to be verified that the method is indeed executed 100.000 times, for this reason input is used. The input is dependent on the output of the previous method and should be the amount of invocations. Due to having a different input every run, the result of the method could not be internally cached. The output of the method is also shown, it can be verified the method has indeed run 100.000 times.

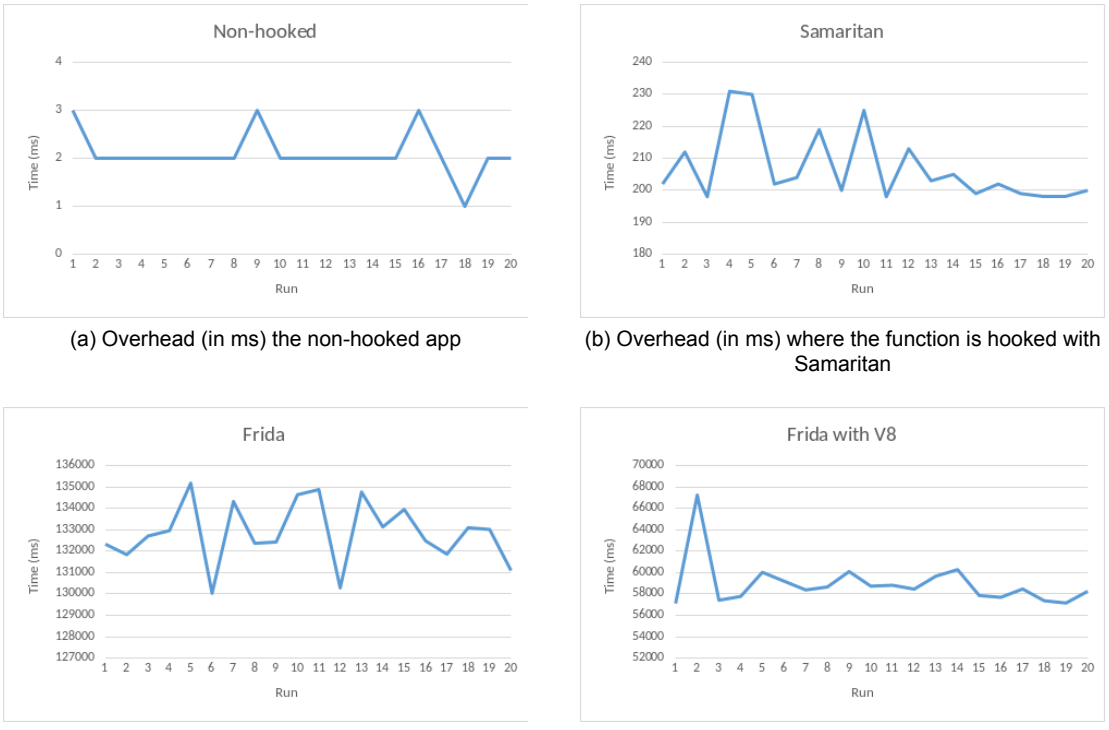
```
public void buttonPressed(View paramView){
    int amount = 0;
    long start = System.currentTimeMillis();
    for(int i = 0; i< 100000;i++){
        amount = doReturn(amount);
    }
```

```
        long end = System.currentTimeMillis();

        long diff = end - start;
        TextView tv = findViewById(R.id.textView);
        tv.setText(diff + " " + amount);

    }

    public int doReturn(int amount){
        return amount + 1;
    }
```

Listing 6.1: Methods used for conducting overhead benchmarking

Frida uses JavaScript and therefore needs a JavaScript engine to expose functions and objects from Java to JavaScript. Standard Duktape [4] is used, but Frida can be configured to work with V8 [5]. For the benchmarking both Frida with Duktape and V8 where tested. When a reference to Frida without a runtime is made, Duktape is used.

The experiment was conducted 20 times, to mitigate for a possible outlier. The results of all 20 runs are shown in Figure 6.2. The average runtime over those 20 runs is shown in Table 6.3. It can be seen that Samaritan introduces an overhead of just under 100 times that of a normal function call. Whilst Frida with V8 has an overhead of 285 times Samaritan, and Frida with Duktape has an overhead of 642 times Samaritan.



(a) Overhead (in ms) the non-hooked app

(b) Overhead (in ms) where the function is hooked with Samaritan

(c) Overhead (in ms) where the function is hooked with Frida

(d) Overhead (in ms) where the function is hooked with Frida (V8)

Figure 6.2: Overhead in the different runs of the program

Recursion

The previous experiment was conducted by having an external function call the `doReturn` method multiple times. This external call could be replaced by a single call and a recursive function. On every invocation of a method a new instance of the function is used. The method will act differently according

[4] https://duktape.org/
[5] https://v8.dev

	Average runtime	Factor compared to non-hooked	Factor compared to Samaritan
Non-hooked	2,1 ms	1x	-
Samaritan	206,9 ms	98,5x	1x
Frida	132.9 s	63271x	642x
Frida V8	58.9 s	28064x	285x

Table 6.3: Average runtime of 100.000 runs for a hooked method

to its input. The results obtained can also be seen as a recursive method calling itself X times. As long as the arguments between the recursive and non-recursive version are the same.

6.4.2. Scalability

The scalability of Frida compared to Samaritan can be observed in multiple ways. First, it was observed how the overhead reacts to different lengths of methods, and different amount of invocations. Secondly, the amount of function calls that could be hooked was investigated.

Fibonacci

In the benchmarking experiment it can be observed that Frida introduces more overhead than Samaritan. In the following experiment the `doReturn` method was replaced with an algorithm for obtaining the Xth number in the fibonacci sequence.

This method was executed N times, so that X * N is 100.000 invocations. The results are shown in Table 6.4. For Frida the amount of overhead increases significantly, whilst for Samaritan this stays almost the same. Samaritan stays under the 4000 ms, which is equivalent to 4 seconds. With Frida this goes up to 260100 ms, meaning 4,5 minutes. Waiting 4,5 minutes for a function call in Android is infeasible and will result in an almost unusable app. During the waiting time pressing the UI, will prompt a pop-up that the app is not responding. Frida with V8 was unable to invoke a function 100.000 times, and this is shown with -.

X x N	Non-Hooked	Samaritan	Frida	Frida V8
10 x 10.000	2536 ms	3295 ms	3765 ms	3929 ms
100 x 1.000	1990 ms	2668 ms	4599 ms	4174 ms
1.000 x 100	1989 ms	3488 ms	6667 ms	5304 ms
10.000 x 10	2342 ms	2906 ms	49950 ms	30591 ms
100.000 x 1	10,15 ms	374 ms	260100 ms	-

Table 6.4: Hooking time of N times obtaining the Xth number of the fibonacci sequence

Maximum function invocations

The amount of hooked function calls that could be executed was investigated. This experiment was conducted with the app from the benchmarking and started with 100.000 invocation. This value was increased with 50.000 after every successfully run. In this experiment no further input was given after starting the function. When using Frida with V8, it stopped working after 150.000 invocations. This increased to 400.000 invocations with the standard Frida. And Samaritan could still handle 1.000.000 invocations after which the experiment was stopped.

When user input was given after starting the experiment, this will decrease the amount of calls. If to much overhead is introduced Android will stop the App as it has become unresponsive. This same setup was used, however the experiment started with 5000 invocations which was increased with 5000 every time.

For Frida both with V8 and Duktape it stopped working after 5000 invocations. Samaritan still succeeded 1.000.000 invocations after which the experiment was stopped.

From this it could be observed that Samaritan can handle a lot more function calls. An application contains more than 5000 methods, and looking at the results it would not be possible to hook all functions. Samaritan can handle over 1 million function invocations and will be usable on a larger scale.

The results of the above experiment are summarized in Table 6.5.

	Samaritan	Frida	Frida V8
Normal	>1.000.000	400.000	150.000
Touch UI	>1.000.000	5000	5000

Table 6.5: Comparison of Samaritan and Frida on hook invocation

6.4.3. Conclusion

It has been shown that Samaritan outperforms Frida in regards of overhead and scalability. Frida is standard a factor 642 slower and with another engine 285 times. Per 100.000 executions Samaritan only introduces 200ms of overhead, which is only 10 times that of a normal function call.

For the amount of function calls that could be hooked, Samaritan outperformed Frida significantly. This shows that Samaritan outperforms Frida in terms of overhead and scalability.

6.5. Finding vulnerabilities

An experiment can be conducted with InsecureBankv2 to show that vulnerabilities can be found with the tool. For this Samaritan is used to hook all functions in the app. A manual analysis on the log file is then used to identify some of the vulnerabilities. Not all vulnerabilities can be found as some are related to the server or are not behaviour related.

6.5.1. Vulnerabilities

Insecure HTTP connections

InsecureBankv2 uses HTTP instead of HTTPS for doing the backend requests. In the log file it can be found that the connections are made with HTTP. An example of such a call is shown in Figure 6.3.

```
org_apache_http_client_methods_HttpPost <init> (Ljava/lang/String;)V http://10.0.0.2:8888/changepassword
Result of: org_apache_http_client_methods_HttpPost <init> (Ljava/lang/String;)V void
```

Figure 6.3: Log entry of a HTTP call

It can be seen that HTTP is used, due to the ip address starting with http://. Inside the log no results were found when searching for SSL or HTTPS.

Using HTTP as opposed to HTTPS is a vulnerability, especially for a bank. With HTTP the connection is not encrypted and anyone on the network can see all information. Sensitive data can be observed by looking at the network data.

Password disclosure

Inside the logs the value for the password can be searched. The password is used in more calls than necessarily needed. It is passed around in the application and is sent unencrypted to the backend. Furthermore, the password is shown in the debug log statements of the application.

6.5.2. Checks

Besides vulnerabilities, countermeasures checks could be detected. These countermeasures are root and emulator detection. These countermeasures are shown, together with an evaluation of the countermeasure usage.

Root detection

Root detection can be found with the keywords from Section 6.3. The InsecureBankv2 app uses two rooting checks.

These calls are:

- **which su** The command which su is executed to see if the command su exist. The su command is used for obtaining root, when a device is rooted. The log entry for this specific call is shown in Figure 6.4.
- **Superuser.apk** It is checked if an apk called Superuser.apk is installed. Superuser is one of the tools for rooting an Android device. The log entry for this specific call is shown in Figure 6.5.

```
Invoked: java_lang_Runtime exec ([Ljava/lang/String;[Ljava/lang/String;Ljava/io/File;)Ljava/lang/Process;
/system/bin/which,su null null
Result of: java_lang_Runtime exec ([Ljava/lang/String;[Ljava/lang/String;Ljava/io/File;)Ljava/lang/Process;
Process[pid=7774]
```

Figure 6.4: The which su log entry

```
Invoked: java_io_File <init> (Ljava/lang/String;)V /system/app/Superuser.apk
Result of: java_io_File <init> (Ljava/lang/String;)V void
```

Figure 6.5: The Superuser.apk log entry

Manual analysis of the code confirmed that these were the only rooting checks.

The rooting checks executed, do not cover all types of rooting checks.

The current root checks have some shortcomings. Not all rooting apps provide the su command to become the root user. Multiple rooting apps exists from which Superuser is one More rooting checks should be used to find more cases of rooting.

Emulator detection

The usage of an emulator can be detected by using keywords. A collection of known checks for anti-emulation is available at [110], and these values are used [6].

The emulation checks of InsecureBankv2 could be found inside the log. The app checks the build values as can be seen in Image Figure 6.6. These are all checks that are executed on build values strings that are not shown in the log. The first check is `Build.FINGERPRINT.startsWith("generic")`. Some checks seem the same, however they check on different build values.

```
Invoked: java_lang_String startsWith (Ljava/lang/String;)Z generic
Result of: java_lang_String startsWith (Ljava/lang/String;)Z false
Invoked: java_lang_String contains (Ljava/lang/CharSequence;)Z unknown
Result of: java_lang_String contains (Ljava/lang/CharSequence;)Z false
Invoked: java_lang_String contains (Ljava/lang/CharSequence;)Z google_sdk
Result of: java_lang_String contains (Ljava/lang/CharSequence;)Z false
Invoked: java_lang_String contains (Ljava/lang/CharSequence;)Z Emulator
Result of: java_lang_String contains (Ljava/lang/CharSequence;)Z false
Invoked: java_lang_String contains (Ljava/lang/CharSequence;)Z Android SDK built for x86
Result of: java_lang_String contains (Ljava/lang/CharSequence;)Z false
Invoked: java_lang_String contains (Ljava/lang/CharSequence;)Z Genymotion
Result of: java_lang_String contains (Ljava/lang/CharSequence;)Z false
Invoked: java_lang_String startsWith (Ljava/lang/String;)Z generic
Result of: java_lang_String startsWith (Ljava/lang/String;)Z false
Invoked: java_lang_String startsWith (Ljava/lang/String;)Z generic
Result of: java_lang_String startsWith (Ljava/lang/String;)Z false
```

Figure 6.6: Emulation detection log entries

It can be seen that only a few values are checked and they are checked for specific values. Different emulators use different build values. Furthermore, there are emulators that mimic the build values of real devices, to circumvent these checks.

It is a vulnerability as these checks can be circumvented and a user could use an emulator. Using an emulator means that the app runs a computer screen.

6.6. Conclusion

To answer the research questions a new tool had to be developed to hook all functions of an Android app. In this section the efficacy of the newly created tool was evaluated. It has been shown that Samaritan can create a call graph that highlights the functions that are executed the most. This can

[6] https://github.com/strazzere/anti-emulator/blob/f2b669a5498076c4071670bd97268971d01ad7ae/AntiEmulator/src/diff/strazzere/anti/emulator/FindEmulator.java

help to point out which methods are of interest for further analysis. Using Samaritan to check for rooting can be done, as the logs provide useful data for manual analysis. It has been shown that Samaritan outperforms Frida in both scalability and overhead, which makes it more usable for large scale evaluations. Lastly, it was shown that Samaritan can find some types of security vulnerabilities, with manual analysis of the logs.

7

Conclusion

In this chapter we will conclude the research and give an overview of the results. Afterwards, the technical difficulties during this thesis are mentioned. Future work with regards to the tool and research will be discussed. Lastly, it is discussed to what extend the tool is future proof.

7.1. Overview of results

At the start of this thesis multiple research questions were defined. In order to answer these questions research was conducted. In this section, the research from the previous chapters is used to answer the questions.

RQ1a: What current research is available that analyses the behaviour of an Android app?

The currently available research was discussed in Chapter 3. It is shown that dynamic analysis is needed to conduct behavioural analysis. Static analysis can be used together with dynamic analysis, but should not be used on its own.

The current research mostly focusses on analysing behaviour on the old runtime of Android. The tooling and methods developed will not work on recent phones. Furthermore, the research was geared towards finding one type of behaviour. This meant that the overall behaviour of an Android app was not captured.

Multiple dynamic analysis techniques were studied. The most used techniques are taint analysis and Dynamic Binary Instrumentation (DBI). With taint analysis only predefined sources can be tracked and this will not aid a generic overview. Depending on the injected code it is possible to create a general overview with DBI. DBI is currently the best approach for doing behaviour analysis.

RQ1b: What tooling is available to conduct behavioural analysis with on Android apps?

In Chapter 3 the current state-of-the-art tooling with research was discussed. These tools could not be used during the research as they were outdated, not public, or did not capture general behaviour. Chapter 4 looks into available tooling for doing the analysis. From this it was concluded that Frida was the only tool that was stable enough to do the analysis with. A proof-of-concept with Frida was made, in order to evaluate its results. It was found that Frida has a limitation on the amount of hooks that can be placed. This invalidated Frida to conduct the analysis as methods need to be hooked to analyse application behaviour.

RQ1: How can the internal behaviour of Android apps be analysed?

The answer to this question is a combination of sub-questions RQ1a and RQ1b.
From RQ1a it was concluded that the internal behaviour can best be studied with DBI. With RQ1b it was found that the currently available tooling did not suffice. The internal behaviour can best be studied with a yet to be developed DBI tool.

RQ2 : How can a tool be build to hook all methods of an Android app to analyse its behaviour?

In order to create the tool some background knowledge was needed which is discussed in Chapter 2. The tool creation process itself was documented in Chapter 5, with the internal workings discussed in Section 5.2.

This resulted in the creation of a tool called Samaritan. With the tool all methods can be hooked, however it requires some manual work. A proof-of-concept of Samaritan shows that the underlying mechanism is able to hook all methods.

RQ2a : What is the efficiency of the newly created tool?

In Chapter 6 the tool build in RQ2a, named Samaritan was evaluated to show how effective it is. Most functions could be hooked when hooking an Android app all together. Samaritan has a small overhead compared to the current state-of-the-art and this results in Samaritan being more scalable. Currently, Samaritan has the limitation that it needs manual input to hook all functions. A few functions could not be hooked yet. However, as a proof-of-concept Samaritan shows that it can hook more than the currently available tooling. The underlying logic should be able to hook al methods, this is most likely due to the hook placement itself.

7.2. Technical challenges

During the creation of the tool, multiple technical challenges were solved. In this section a summary of these technical challenges is given, together with their solutions.

7.2.1. Dynamic

For a hooking framework to hook all functions off an Android app, those functions have to be dynamically hooked. No hooking tool uses dynamic hook creation. For Frida a script with dynamic hook creation can be made, this is possible because of JavaScript. A method for dynamic hook creation had to be designed from scratch.

Assembly

The current solution for dynamic hook creation is using assembly as a middle layer. For every method an assembly function is pre-generated. For the assembly to communicate with the actual framework, only a few generic functions are needed.

Opcodes

In order to have a more scalable solution the pre-generation step has to be removed. For this opcodes can be used, which can be generated on the fly and can be executed inside the framework. The opcodes will replace the assembly code, and will need the same generic functions. This has not yet been implemented in the current version of the framework.

More on this challenge can be read in Section 5.3.1.

7.2.2. Injection

To inject the shared library inside the target app a few technical challenges had to be solved.

ASLR

In order to inject code inside the target app, ASLR had to be circumvented. This could be done, because a function has a predetermined offset inside a library. The local offset could then be used, to calculate the remote offset.

LD_PRELOAD

In order for a library to be injected and to run on a system it has to be loaded by the dynamic linker. LD_PRELOAD is used to work around this problem.

An elaborate explanation of the whole injection process can be found in Section 5.2.1.

7.2.3. Hooking

Building a hooking tool is not a trivial task and the internals of Android had to be studied. The biggest challenge was figuring out how a function could be hooked, while keeping the original function intact. For this the ArtMethod struct was studied, mostly entry_point_from_interpreter_ and entry_point_from _quick_compiled_code_. For both entry points it had to be figured out which values had to used and how these values could be obtained.

7.3. Future work

In this research project there is still future work. In this section the future work with regards to the tool and research itself is discussed.

7.3.1. Samaritan

Although Samaritan can currently hook most methods there is quite some future work to be done. This future work would enable the usage of Samaritan for more behavioural research. The future work of Samaritan is discussed.

64 bit

Currently, only 32 bit ARM is supported. There are a few places that have to be adapted to support 64 bit.

- **Injection** The injection is done with ptrace. Although ptrace has the needed functionality on both 32 and 64 bit, the arguments differ significantly. Only arguments have to be substituted and this has been done by others before, this is an easy improvement.

- **Native hooks** The native framework has to be totally adapted to support 64 bit hooking. The same logic applies to 32 and 64 bit. Only implementation details have to be changed.

- **Assembly** In order to hook all the methods assembly code was generated. Assembly code for 32 and 64 bit are different and a new script has to be created to generate 64 bit assembly.

To support 64 bit self placed hooks, only the injection process has to be changed. For native hooks and hooking all Java functions, more adaptions have to be made. The logic for adapting both has already been figured out, only some work has to be spent to do the implementation. Besides the part mentioned above, no hard coded values are used with regard to 32 bit, in order to make it expandable. It is feasible to support 64 bit in the future.

Obtaining native address

In order to hook a function in C/C++ its memory address needs to be obtained. Currently, this memory address has to be manually obtained with static analysis. In order to hook all native functions, it would be useful if no manual work was needed. How this can be done has to be investigated, as no research has been done into this topic.

Android 6 and above

In order to support Android 6 and above a few small changes have to be done. Java hooks currently work with a call to dlopen, which has been blocked since Android 7. There are solutions available on the Internet to solve this, and this was not implemented due to time constraints. It is however very feasible to solve this problem and support higher version of Android. The framework itself was kept as generic as possible to require little modification for supporting newer Android versions.

On start-up

Currently, the shared library is attached by the user. This causes a delay, where the start-up of the app cannot be included in the analysis. It should be investigated if and how it is possible to start the app with the shared library already attached. This way the starting process can be included.

Scripting language
Currently, hooks are placed with C++. Inside a C++ program it is possible to invoke Lua or Python. For user placed hooks there should be the possibility to write to hooks in Lua or Python for usability. If overhead is a problem or someone prefers C++, it should still be possible to use C++. This is possible due to Lua/Python integration with C/C++. Lua support was already tested in Samaritan and this integration is working. Only the bindings have to be created to support this. For a proof-of-concept this is not needed and this was not implemented further. It is proven that Lua works with Samaritan, making this integration not difficult. When Python is the scripting language of choice a port to Android has to be found first.

Multi-threading support
The native part of Samaritan has no support for multi-threading. This support is broken by the fact that an Android application has multiple native helper threads and some of them crash the tool. The problem with these has to be investigated and solved. Multi-threading support for native code is already included.

Automatize
A beginning was made to dynamically hook all functions. This work should be continued in order to automatically hook an Android app. An automated process can be used for UI interaction. This would result in the automatic testing of an Android app.

All Java functions
There are some Java functions that can currently not be correctly hooked by Samaritan. This only occurs during the hooking of the whole app, and therefore it could be a side-effect of the implementation. First, the dynamic hook creation has to be created as this might mitigate the problem. Afterwards, it has to be investigated to what extend this solves the current problems.

7.3.2. Research
Not only future work regarding the tool, but also to the research itself is possible. In this section future work regarding the research will be discussed.

After implementing the future work of the tool, full scale analysis of Android apps can be conducted. With the improved version new research can be done, that focuses on finding behaviour. Applications can be tested to answer to research question *Can X be found with behavioural analysis of Android apps?*. Where X could be malicious behaviour, dead code or countermeasures.

Detecting behaviour is step one, acting upon it is step two. A next step should be investigating how the behaviour of an app can be changed. This could result in automatically evading countermeasures. For this¸ real time analysis needs to be conducted in order to act upon this behaviour.

Other research can be done in finding malicious calls and changing the outcome to stop the communication.

Fuzzing
Samaritan will only display the executed function calls. If an execution path is not used, the function calls related will not be visible. Adding a fuzzer increases the changes of executing unused execution paths. Furthermore, different user inputs could be tried without manual effort. Currently, there are no good Android fuzzers available. Mostly the UI/Application Exerciser Monkey from Android [1] is used. This is a simple clicker that has difficulties with running the app. For example, it has difficulties exiting activities when no back button is available in the app. Whenever a good fuzzer is available this should be combined with the tool, to get an automatic overview of the system.

[1] https://developer.android.com/studio/test/monkey

A

Adb functionality

In this appendix the ADB functionality that is used during this thesis is explained. There are a lot of commands, and not all of them will be discussed.

A.1. Starting the server

With *adb start-server* the adb server is started. It is not required to use this command, as its executed in the background when a command is entered without starting the server first. This command is mostly used implicitly.

A.2. List devices

With *adb devices* the currently connected devices can be listed. The ids of the devices are also displayed, the id is added to a command when a client is connected to multiple servers.

A.3. Shell

With adb it is possible to execute shell commands or open a shell connection. To send only one shell command *adb shell <command>* is used, to execute that specific command. When running *adb shell* without any arguments a shell is opened in which multiple commands can be executed. This shell functions as a normal shell and can be upgraded to root if the device is rooted.

A.4. Root

If the device has been rooted a user can send commands to the server as a root user. For this the command *adb root* is used, which restarts the daemon as root. If the root permissions are not needed any more, the daemon can be restarted as normal user with *adb unroot*. When root access is only needed inside a shell, a normal upgrade to root can be done in the shell itself by executing the command for root access.

A.5. Viewing logs

Android has its own logging tool called logcat. It can be used to view logs, which can be invoked by using *adb logcat*. With logcat logs can for example be filtered on tag or the log level and output format can be changed. The user can place logs in both Java and native code. Besides the user placed logs, logcat also contains system logs.

A.6. File sharing

It is possible to upload a file from the computer to the Android device and vice versa. Uploading a file from the computer to the Android device is done via *adb push <local location> <remote location>*. Likewise, a file can be downloaded with *adb pull <remote location> <local location>*, where for the current directory the local location can be omitted.

A.7. Installing an app

For Android the package file format APK is used. In order to install a custom app the command *adb install* <*location of apk*> is used. This will first upload the application to the Android device and will then install it.

B

Frida script

```
1  // const Java = require('frida-java');
2
3  //Source: nccgroup/house
4  function enumDexClasses(apk_path) {
5      var DexFile = Java.use("dalvik.system.DexFile");
6      var df = DexFile.$new(apk_path);
7      var en = df.entries()
8
9      var dexClasses = []
10     var i = 0
11     while (en.hasMoreElements()) {
12         i += 1
13         var a = en.nextElement()
14         if (i != 666) {
15             dexClasses.push(a);
16         }
17     }
18     send("dex classes: " + i)
19
20     return dexClasses;
21 }
22
23 //Inspiration from: nccgroup/house
24 function enumAllClasses() {
25     var allClasses = [];
26     var classes = Java.enumerateLoadedClassesSync();
27     send(classes.length)
28     classes.forEach(function (aClass) {
29         try {
30             var className = aClass.replace(/\//g, ".");
31         } catch (err) { }
32         //The 'classes' byte, char, long etc. are enumerated but are no real class
33         if (className.includes(".") && !className.startsWith('[')) {
34             allClasses.push(className);
35         }
36     });
37     return allClasses;
38 }
39 function enumAllClassesCounter() {
40     var allClasses = [];
41     var classes = Java.enumerateLoadedClassesSync();
42     var counter = 0
43     var start = 750
44     var limit = 800
45     send(classes.length)
46     classes.forEach(function (aClass) {
47         try {
48             var className = aClass.replace(/\//g, ".");
49         } catch (err) { }
50         //The 'classes' byte, char, long etc. are enumerated but are no real class
```

```
51          if (counter < start) {
52              counter++
53          }
54          else if (counter > limit) {
55              return
56          }
57          else if (className.includes(".") && !className.startsWith('[')) {
58              counter++
59              allClasses.push(className);
60          }
61
62      });
63      send(counter)
64      return allClasses;
65  }
66
67  function getMethodName(m) {
68      // console.log(m)
69      var openB = m.indexOf('(');
70      var closeB = m.indexOf(')')
71
72      var args = m.substring(openB + 1, closeB)
73      m = m.substring(0, openB)
74
75      var dot = m.lastIndexOf('.');
76      var meth = m.substring(dot + 1, openB)
77      // console.log(meth)
78      return { 'meth': meth, 'args': args }
79
80  }
81
82  //Inspiration from: nccgroup/house
83  function enumMethods(targetClass) {
84      var hook = Java.use(targetClass);
85      var temp = hook.class.getDeclaredMethods();
86
87      var ownMethods = [];
88      for (var i in temp) {
89          var m = getMethodName(temp[i].toString())
90          if (!m['meth'].startsWith("-")) {
91              // if (!m['meth'].includes("$") && !m['meth'].startsWith("-")) {
92              ownMethods.push(m)
93          }
94      }
95
96      hook.$dispose;
97
98      return ownMethods;
99  }
100 setImmediate(function () {
101     Java.performNow(function () {
102         var classes = enumAllClassesCounter()
103         // var classes = enumDexClasses("/data/app/grandctf.brs.com.brs_ctf-1/base.apk")
104         for (var c in classes) {
105             var className = classes[c].toString();
106             // send(className)
107             if (className.includes("$")) {// || className.startsWith('android')){// ||
                    className.startsWith('libcore') || className.startsWith('com.android')) {//||
                     (!className.startsWith('java.lang.Runtime') && !className.includes('java.io.
                    File') && !className.includes('brs') && !className.includes('beer'))) {
108                 // if (className != "java.security.cert.CertPath") {
109                 continue;
110             }
111             var class_hook = eval('Java.use("' + className + '")');
112             var methods = enumMethods(className);
113
114             for (var method_i in methods) {
115                 if (!methods.hasOwnProperty(method_i)) {
116                     //We continue if the key is not a real property of the object
117                     continue;
118                 }
```

```
119                 var method = methods[method_i]
120                 var methodName = method['meth']
121                 // send(className + "." + methodName);
122                 if (className + "." + methodName != 'java.lang.Class.isInstance') {
123                     var overloads = class_hook[methodName].overloads;
124                     for (var ovl_i in overloads) {
125                         //Function for scoping
126                         (function (currentMethod, currentClass) {
127                             overloads[ovl_i].implementation = function () {
128                                 var currentMethodName = currentMethod['meth']
129                                 var ret = eval('this.' + currentMethodName + '.apply(this,
                                        arguments)')
130                                 // send(lastcall + " -> " + currentMethodName)
131                                 // lastcall = currentMethodName
132                                 send(currentClass + '.' + currentMethodName + '(' +
                                        currentMethod['args'] + ')  args: ' + JSON.stringify(
                                        arguments) + ' ret: ' + JSON.stringify(ret));
133                                 return ret;
134                             }
135                         })(method, className)
136                     }
137                 }
138             }
139         }
140         send("Hooks placed")
141
142         //This should mitigate the pthread problem
143         //Source: https://enovella.github.io/android/reverse/2017/05/20/android-owasp-
                crackmes-level-3.html
144         var p_pthread_create = Module.findExportByName("libc.so", "pthread_create"); //libc
                <-- standard C-lib so this should work for all apps.
145         var pthread_create = new NativeFunction(p_pthread_create, "int", ["pointer", "pointer
                ", "pointer", "pointer"]);
146         // send("NativeFunction pthread_create() replaced @ " + pthread_create);
147
148         Interceptor.replace(p_pthread_create, new NativeCallback(function (ptr0, ptr1, ptr2,
                ptr3) {
149             if (ptr1.isNull() && ptr3.isNull()) {
150                 send("Native pthread check observed");
151             } else {
152                 pthread_create(ptr0, ptr1, ptr2, ptr3);
153             }
154
155         }, "int", ["pointer", "pointer", "pointer", "pointer"]));
156
157         send("Hooks placed")
158     });
159 });
```

Listing B.1: Frida script used for the PoC

C

Samaritan native script

All code for placing a function can be found in Listing C.1, where *hook_check* is the method to be hooked. The code contains code comments to explain the lines.

```
// Function definition for trampoline
jstring (*old_check)(JNIEnv *, jobject, jint) = NULL;

//Memory address of function
uint32_t check_offset = 0x45b1;

// Replaced function for method check
jstring new_hook_check(JNIEnv *jni, jobject self, jint i)
{
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "Invoked check with arguments %d
        \n", i);
    jstring c = old_check(jni, self, i);
    const char *str = jni->GetStringUTFChars(c, 0);
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "Return of function check is %s \
        n", str);
    env->ReleaseStringUTFChars(str, java_str);
    return c;
}

// Method for creating a hook for check
void hook_check()
{
    const char *name = "libnative-lib.so";
    //Function in native part to obtain the base address of the library
    uint32_t base_address = get_addr_range(name).base;
    uint32_t offset = base_address + check_offset;
    __android_log_print(ANDROID_LOG_DEBUG, "samaritan", "check_offset address %u\n",
        offset);
    //Function in native part, to hook the function
    hook(offset, (uint32_t)new_hook_check, (uint32_t **)&old_check);
}
```

Listing C.1: Code to place a native hook

D

Assembly code

The following is the assembly code that functions as a bridge between the function call and the framework itself. The code is for the getRand function, which takes no arguments and returns an Integer. The JNI adds two arguments to the function call and these arguments are also added to the stack.

```
1   .arch armv7-a
2   .fpu softvfp
3   .section  .rodata
4   .align  2
5  .LC0:
6   .ascii  "com_example_hellojni_HelloJni\000"
7   .align  2
8  .LC1:
9   .ascii  "getRand\000"
10  .align  2
11 .LC2:
12  .ascii  "()I\000"
13  .align  2
14  .text
15  .align  2
16  .global j_new_com_example_hellojni_HelloJni_getRand_3367
17  .type j_new_com_example_hellojni_HelloJni_getRand_3367, %function
18 j_new_com_example_hellojni_HelloJni_getRand_3367:
19  @ args = 4, pretend = 0, frame = 48
20  @ frame_needed = 1, uses_anonymous_args = 0
21  stmfd sp!, {fp, lr}             @ Full descending stack
22  add fp, sp, #4                  @ Increment fp, to make room for local vars
23  sub sp, sp, #48                 @ Make space free on the stack
24  str r0, [fp, #-40]              @ store arg:1 (JNIEnv* env)
25  str r1, [fp, #-32]              @ store arg:2 (Jobject self)
26  sub r0, fp, #40                 @ Get pointer to beginning of struct
27  str r0, [sp]                    @ Store as first arg on stack
28  ldr r3, .L5
29 .LPIC0:
30  add r3, pc, r3                  @ Obtain classname
31  str r3, [fp, #-16]              @ Store classname
32  mov r3, #3367                   @ Obtain id
33  str r3, [fp, #-24]              @ Store id
34  ldr r3, .L5+4
35 .LPIC1:
36  add r3, pc, r3                  @ Obtain methodName
37  str r3, [fp, #-20]              @ Store methodName
38  ldr r3, .L5+8
39 .LPIC2:
40  add r3, pc, r3                  @ Obtain signature
41  str r3, [fp, #-12]              @ Store signature
42  sub r3, fp, #24                 @ Get pointer to beginning of struct
43  ldmia r3, {r0, r1, r2, r3}
44  bl  _Z14genericCallInt8hookInfo8hookArgs(PLT)
45  sub sp, fp, #4                  @ Restore stack pointer
46  ldmfd sp!, {fp, pc}             @ Restore registers and return
```

```
47 .L5:
48   .word .LC0-(.LPIC0+8)
49   .word .LC1-(.LPIC1+8)
50   .word .LC2-(.LPIC2+8)
51   .size j_new_com_example_hellojni_HelloJni_getRand_3367, .-
         j_new_com_example_hellojni_HelloJni_getRand_3367
```

Listing D.1: Assembly code for the function getRand

Glossary

dex Dalvik Executable format files, which is an intermediate step between Java bytecode and machine code in Android. 5, 11

DOT A language for describing graphs. 25, 32

double word A double word are two adjacent words.. 31

instance method An instance method is a Java method which requires an object of its class to be created before the method can be invoked. 11

lambda A function definition that is not bound to an identifier, they can be passed as a variable.. 31

metaprogramming Metaprogramming results in a program that can manipulate programs including itself.. 19

MethodID Java Native Interface method identifier. 11

native code Native code is code that is compiled to run on a specific type of processor.. 9

OpenGL A non programming language specific library for rendering 2D and 3D vectors. 7

smali The assembly language used for Android. 17, 19

static method A static method is a Java method which can be invoked without first creating a object of its class. 11

word A word is the standard length of data for a specific processor.. 31

YARA A pattern matching tool, mostly used for malware research. 19

Acronyms

ADB Android Debug Bridge. 12, 47

AOT Ahead Of Time. 7

API Application Programming Interface. 5, 7, 24

APK Android Package. 5, 48

app application. 1, 5, 8, 16

ART Android Run Time. 5, 8, 17, 18

ASLR Address Space Layout Randomization. 8, 9, 26, 44

DBI Dynamic Binary Instrumentation. 3, 16, 17, 19, 20, 22, 43

DVM Dalvik Virtual Machine. 5, 7

HAL Hardware Abstraction Layer. 5

HCE Host Card Emulation. 18, 19

JavaVM Java Virtual Machine. 9

JIT Just In Time. 7

JNI Java Native Interface. 9, 11, 55

JNIEnv Java Native Interface Environment. 9

JVM Java Virtual Machine. 7, 9, 13

OS Operating System. 17, 35

PHA Potential Harmful App. 1, 2

PoC Proof of Concept. 23

SELinux Security-Enhanced Linux. 8, 26

UI User Interface. 7, 33, 38

VM Virtual Machine. 5, 8, 9, 19, 29

Bibliography

[1] Vishal Mishra. Android hooking: Writing xposed modules, April 2016. URL http://infosec.vishalmishra.in/2016/04/android-hooking-writing-xposed-modules.html.

[2] Android Open Source Project. Android platform architecture, . URL https://developer.android.com/guide/platform.

[3] Sheng Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[4] Jurriaan Bremer. x86 api hooking demystified, July 2012. URL http://jbremer.org/x86-api-hooking-demystified.

[5] statcounter. Desktop vs mobile vs tablet vs console market share worldwide, September 2019. URL https://gs.statcounter.com/platform-market-share.

[6] Melissa Chau and Ryan Reith. Smartphone market share, 2019. URL https://www.idc.com/promo/smartphone-market-share/os.

[7] statcounter. Mobile operating system market share worldwide, September 2019. URL https://gs.statcounter.com/os-market-share/mobile/worldwide.

[8] Nokia. Nokia threat intelligence report – 2017. Technical report, 2017. URL https://nokiamob.net/wp-content/uploads/2017/11/Nokia-Security-Report.pdf.

[9] Andrew Ahn. How we fought bad apps and malicious developers in 2017, January 2018. URL https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html.

[10] Paul D Miller and Svitlana Matviyenko. *The imaginary app*. The MIT Press, 2014.

[11] Jillian D'Onfro. Google is missing out on billions of dollars by not having an app store in china, new data shows, January 2018. URL https://www.cnbc.com/2018/01/17/google-misses-out-on-billions-in-china.html.

[12] Google. Malicious behaviour, . URL https://play.google.com/intl/en-GB/about/privacy-security-deception/malicious-behavior/.

[13] Google. Potentially harmful applications (phas), . URL https://developers.google.com/android/play-protect/potentially-harmful-applications.

[14] Android. Android security & privacy 2018 year in review, March 2019. URL https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf.

[15] Google. Google play protect: Securing 2 billion users daily, . URL https://www.android.com/intl/en_en/play-protect/.

[16] Rahul Mishra and Tom Watkins. Google play protect in 2018: New updates to keep android users secure, February 2019. URL https://android-developers.googleblog.com/2019/02/google-play-protect-in-2018-new-updates.html.

[17] AV-TEST. Test google play protect 13.9 for android (191011), April 2019. URL https://www.av-test.org/en/antivirus/mobile-devices/android/march-2019/google-play-protect-13.9-191011/.

[18] ISO 27032:2012. Guidelines for cyber security. Standard, International Organization for Standardization, Geneva, CH, July 2012.

[19] Christian Doerr. *Network Security in Theory and Practice*. 2018.

[20] Ankit Sinhal. Closer look at android runtime: Dvm vs art, April 2017. URL https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924.

[21] Tam Doan. Virtual machine in android: Everything you need to know, August 2019. URL https://medium.com/@nhoxbypass/virtual-machine-in-android-everything-you-need-to-know-9ec695f7313b.

[22] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4(8), 2010.

[23] Android Open Source Project. Android runtime (art) and dalvik, . URL https://source.android.com/devices/tech/dalvik.

[24] Nourdin Aït El Mehdi. Analyzing the resilience of modern smartphones against fault injection attacks. Master's thesis, Delft University of Technology, The Netherlands, 2019.

[25] Ayusch Jain. Android internals: Art vs dvm in android. URL https://ayusch.com/art-vs-dvm-in-android/.

[26] Android Open Source Project. Android application fundamentals, . URL https://developer.android.com/guide/components/fundamentals.html.

[27] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.

[28] Rasmus Nørgaard. The zygote process, March 2015. URL https://medium.com/masters-on-mobile/the-zygote-process-a5d4fc3503db.

[29] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[30] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *Ndss*, volume 310, pages 20–38, 2013.

[31] Android Open Source Project. Security-enhanced linux in android, . URL https://source.android.com/security/selinux.

[32] Di Shen. Defeating samsung knox with zero privilege. *BlackHat USA*, 2017.

[33] Hang Zhang, Dongdong She, and Zhiyun Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1093–1104. ACM, 2015.

[34] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14. ACM, 2015.

[35] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[36] Nancy Hua, James Brandon Koppel, and Jeremy Nelson Orlow. Enhanced code callback, November 2016. US Patent 9,483,283.

[37] Geonbae Na, Jongsu Lim, Sunjun Lee, and Jeong Hyun Yi. Mobile code anti-reversing scheme based on bytecode trapping in art. *Sensors*, 19(11):2625, 2019.

[38] Oracle. Jni types and data structures, . URL https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html.

[39] Oracle. Design overview, . URL https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html.

[40] Nancy Hua, James Brandon Koppel, and Jeremy Nelson Orlow. Enhanced code callback, January 2017. US Patent App. 15/282,290.

[41] Oracle. Retrieving and parsing method modifiers, . URL https://docs.oracle.com/javase/tutorial/reflect/member/methodModifiers.html.

[42] Android Open Source Project. modifiers.h, . URL https://android.googlesource.com/platform/art/+/refs/heads/marshmallow-release/runtime/modifiers.h.

[43] Android Open Source Project. Android debug bridge (adb), . URL https://developer.android.com/studio/command-line/adb.

[44] Maxim Shafirov. Kotlin on android. now official, May 2017. URL https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/.

[45] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *International workshop on information hiding*, pages 270–284. Springer, 2011.

[46] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[47] Thoms Ball. The concept of dynamic analysis. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 216–234. Springer-Verlag, 1999.

[48] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[49] Daniel Mercier, Aziem Chawdhary, and Richard Jones. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501. IEEE, 2017.

[50] Filippo Alberto Brandolini. Hooking java methods and native functions to enhance android applications security. Master's thesis, University of Bologna, Italy, 2016.

[51] Collin Mulliner. adbi - the android dynamic binary instrumentation toolkit. URL https://github.com/crmulliner/adbi.

[52] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[53] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[54] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[55] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[56] Riyadh Mahmood, Naeem Esfahani, Thabet Kacem, Nariman Mirzaei, Sam Malek, and Angelos Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 22–28. IEEE press, 2012.

[57] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou. Behavioral analysis of android applications using automated instrumentation. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 182–187. IEEE, 2013.

[58] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.

[59] Alexandru Blanda. Fuzzing android: a recipe for uncovering vulnerabilities inside system components in android. *BlackHat EU*, 2015, 2015.

[60] Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.

[61] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*, pages 3–17. IEEE, 2014.

[62] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001*, 2014.

[63] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[64] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[65] Patrik Lantz. An android application sandbox for dynamic analysis. Master's thesis, Lund university, Sweden, 2011.

[66] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. Dynalog: An automated dynamic analysis framework for characterizing android applications. In *Cyber Security And Protection Of Digital Services (Cyber Security), 2016 International Conference On*, pages 1–8. IEEE, 2016.

[67] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[68] Gérard Wagener, Alexandre Dulaunoy, et al. Malware behaviour analysis. *Journal in computer virology*, 4(4):279–287, 2008.

[69] Xing Wang, Wei Wang, Yongzhong He, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Characterizing android apps' behavior for effective detection of malapps at large scale. *Future Generation Computer Systems*, 75:30–45, 2017.

[70] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[71] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[72] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[73] Riscure. Analyzing the security of cloud-based payment apps on android. Technical report, 2018. URL `https://www.riscure.com/publication/analyzing-security-cloud-based-payment-apps-android/`.

[74] Carsten Willems and Felix C Freiling. Reverse code engineering—state of the art and countermeasures. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 54(2):53–63, 2012.

[75] Ratinder Kaur, Yan Li, Junaid Iqbal, Hugo Gonzalez, and Natalia Stakhanova. A security assessment of hce-nfc enabled e-wallet banking android apps. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 492–497. IEEE, 2018.

[76] Ajin Abraham, Dominik Schlecht, Magaofei, Matan Dobrushin, and Vincent Nadal. Mobile security framework (mobsf), 2015. URL `https://github.com/MobSF/Mobile-Security-Framework-MobSF`.

[77] Long Nguyen-Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung. Android rooting: An arms race between evasion and detection. *Security and Communication Networks*, 2017, 2017.

[78] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21, 2018.

[79] Rowland Yu. Android packers: facing the challenges, building solutions. In *Proceedings of the 24th Virus Bulletin International Conference*, 2014.

[80] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 359–381. Springer, 2015.

[81] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 358–369. IEEE, 2017.

[82] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51: 16–31, 2015.

[83] Ashish Anand. Securing android code using white box cryptography and obfuscation technique. *International journal of computer science and mobile computing*, 4(4):347–352, 2015.

[84] Hidde Boomsma. Dead code elimination for web applications written in dynamic languages. Master's thesis, Delft University of Technology, The Netherlands, 2012.

[85] Yih-Farn Chen, Emden R Gansner, and Eleftherios Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24 (9):682–694, 1998.

[86] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679. IEEE, 2015.

[87] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518. IEEE, 2017.

[88] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[89] Ole André Vadla Ravnås. Frida. URL https://www.frida.re/.

[90] Romain Thomas. How to use frida on a non-rooted device. URL https://lief.quarkslab.com/doc/latest/tutorials/09_frida_lief.html.

[91] rovo89. Development tutorial, April 2016. URL https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[92] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. Lumus: Dynamically uncovering evasive android applications. In *International Conference on Information Security*, pages 47–66. Springer, 2018.

[93] rovo89. Xposed for lollipop/marshmallow/nougat/oreo [v90-beta3, 2018/01/29], February 2015. URL https://forum.xda-developers.com/showthread.php?t=3034811.

[94] LLC SaurikIT. Cydia substrate. URL http://www.cydiasubstrate.com/.

[95] Derek Bruening. Dynamorio, dynamic instrumentation tool platform. URL http://www.dynamorio.org/.

[96] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.

[97] Oliver Schranz. Big news: Beta, droidcon berlin & black hat usa, June 2018. URL https://artist.cispa.saarland/2018/06/22/beta-and-talks.html.

[98] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 481–495. IEEE, 2017.

[99] Valerio Costamagna and Cong Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *IMPS@ ESSoS*, pages 20–28, 2016.

[100] Michael Haardt, Mike Coleman, Denys Vlasenko, and Michael Kerrisk. ptrace. URL http://man7.org/linux/man-pages/man2/ptrace.2.html.

[101] Alexandr Fadeev. Shared library injection on android 8.0, August 2018. URL https://fadeevab.com/shared-library-injection-on-android-8/.

[102] Inc. Gentoo Foundation. Selinux/labels. URL https://wiki.gentoo.org/wiki/SELinux/Labels.

[103] Shǔ Chén. Shared library injection in android, March 2016. URL https://shunix.com/shared-library-injection-in-android/.

[104] Simone Margaritelli. Dynamically inject a shared library into a running process on android/arm, may 2015. URL https://www.evilsocket.net/2015/05/01/dynamically-inject-a-shared-library-into-a-running-process-on-androidarm/.

[105] ele7enxxh. Android-inline-hook. URL https://github.com/ele7enxxh/Android-Inline-Hook.

[106] Scott Alexander-Bown and Mat Rollings. Rootbeer - simple to use root checking android library and sample app. URL https://github.com/scottyab/rootbeer.

[107] Dinesh Shetty. Android-insecurebankv2, 2019. URL https://github.com/dineshshetty/Android-InsecureBankv2.

[108] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[109] Anthony Desnos. Androguard - reverse engineering, malware and goodware analysis of android applications ... and more (ninja !). URL `https://github.com/androguard/androguard`.

[110] Tim Strazzere. Dex education 201: anti-emulation. *HITCON2013*, 2013.