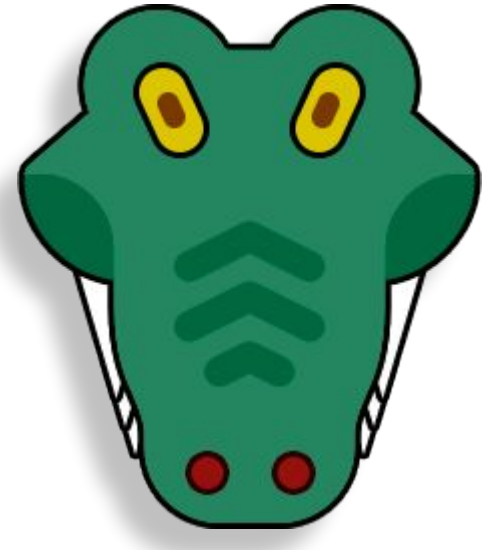
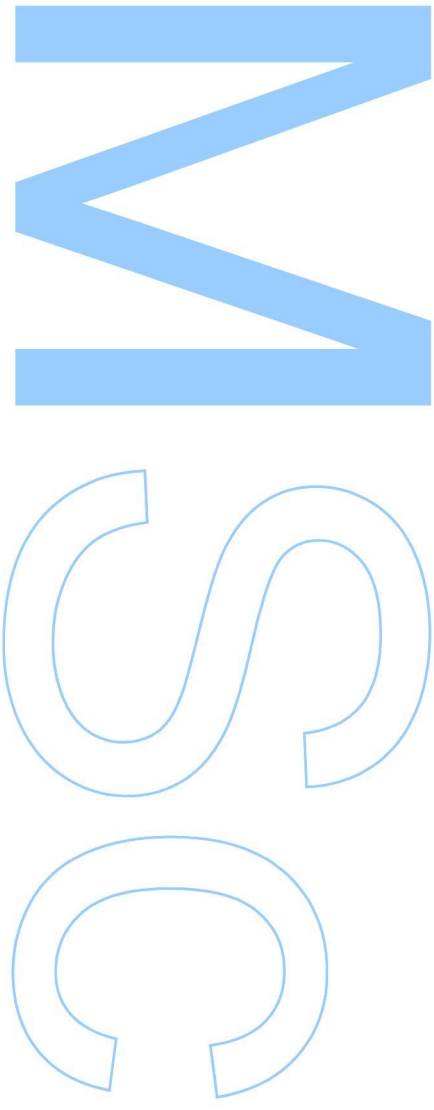


Android Security by Introspection

João Vasco Bispo Estrela
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2019

Orientador
Rolando da Silva Martins
Professor Auxiliar
Faculdade de Ciências da Universidade do Porto

Coorientador
João Miguel Maia Soares de Resende
Professor Assistente Convidado
Faculdade de Ciências da Universidade do Porto



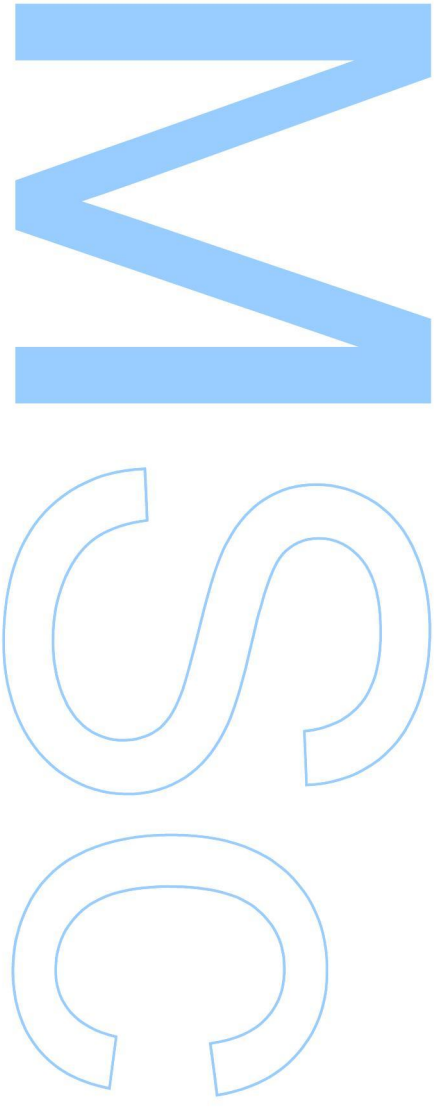
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Abstract

Smartphones have become a necessity in everyday life for most people. These devices are a huge repository of personal data and thus become a valuable target for information gathering.

Access to this kind of personal data can be used to understand an individual's behaviour, allowing for the establishment of associated profiles. These profiles can be used to manipulate individuals in various contexts, for example, what to buy or what to believe.

The increasing awareness of the importance of this data requires new solutions to protect it and we propose a new security system for the Android operating system, one of the most popular platforms where this data is collected. Our system is focused on providing any user with the ability to change the behaviour of applications installed on their devices so that data collection can be monitored and modified to better suit the user policies.

Resumo

Os smartphones tornaram-se uma necessidade na vida cotidiana para a maioria das pessoas. Estes dispositivos são um enorme repositório de dados pessoais e, portanto, tornam-se um alvo valioso para a recolha de informações.

O acesso a esse tipo de dados pessoais pode ser usado para entender o comportamento de um indivíduo, permitindo o estabelecimento de perfis associados. Esses perfis podem ser usados para manipular indivíduos em vários contextos, por exemplo, o que comprar ou em que acreditar.

O aumento da conscientização sobre a importância desses dados exige novas soluções para protegê-los e propomos um novo sistema de segurança para o sistema operacional Android, uma das plataformas mais populares onde esses dados são coletados. Nosso sistema está focado em fornecer a qualquer utilizador a capacidade de alterar o comportamento das aplicações instaladas mos seus dispositivos, para que a coleta de dados possa ser monitorada e modificada para melhor se adequar às políticas do utilizador.

Acknowledgments

I want like to thank my thesis advisor, Rolando Martins for the opportunity to work in this project and for all suggestions that improved my thesis.

I also want like my thesis co-advisor, João Resende for all the guidance, recommendations and support given during the development of this thesis.

To Rafael Almeida and Duarte Figueiredo for listening me while rambling about my thesis and giving moments of so much needed distraction.

I want to thank my best friends for their patience and support during some difficult hours.

Special thank to my parents, Vasco and Anabela, because without them would never be who I'm and where I'm today.

Finally, last but not the least, my beloved Tânia Carvalho, for all the love, patience, support, encouragement, time and all of that stuff.

Dedicated to
my parents and Tânia

Contents

List of Tables

List of Figures

Listings

Acronyms

AIDL Android Interface Definition Language

AOP Aspect-Oriented Programming

API Application Programming Interface

APK Android Package

CPU Central Processing Unit

CSV Comma-separated values

GDPR General Data Protection Regulation

GCP Google Cloud Platform

GUI Graphical User Interface

IPC Inter-Process Communication

HAL Hardware Abstraction Layer

HTTP Hyper Text Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IPC Inter-Process Communication

JDK Java Development Kit

JNI Java Native Interface

JSON JavaScript Object Notation

NDK Native Development Kit

OS Operating System

PSTN Public Switched Telephone Network

RAM Random Access Memory

REST Representational State Transfer

RPC Remote Procedure Call

SaaS Software as a Service

SDK Software Development Kit

SE Standard Edition

SMS Short Message Service

SQL Structured Query Language

SSL Secure Socket Layer

UI User Interface

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Chapter 1

Introduction

The innovations in the field of microprocessors have lead to the creation of devices with large computing power that are able to be carried by anyone, anywhere. One example of these devices is the mobile phones that started with a constrained number of functions in the early stages. At the start only offering voice such as in the Public Switched Telephone Network (PSTN), and later with messages with the purpose of simplifying communications.

In the early 2000's the smartphone introduced a new paradigm (Figure 1.1). This new iteration of mobile devices brought hardware that enabled many the usage of today features like Internet connectivity, cameras and various sensors, such as GPS. By 2007 new mobile operating systems (Android, IOS, Windows Phone) started to emerge to make the use of this modern hardware and quickly replaced the old ones. These innovations made easy to perform many tasks that previously required a personal computer but came at the cost of collecting user information to execute them.

At this time one of the most controversial problems of today started to arise, the lack of privacy. These modern devices are capable of performing many tasks that before were thought impossible on such small and mobile equipment and became deposits of personal data that hold high value in today's markets [34]. This data is extremely valuable as it enables the creation of advertisement profiles for targeting advertisements or to discriminate persons based on information that otherwise would be piratically impossible to know, like, medical records, sexual orientation or even religious beliefs. The sources for the retrieval of this data include text mining from Short Message Service (SMS) or audio mining in recorded clips from the microphone.

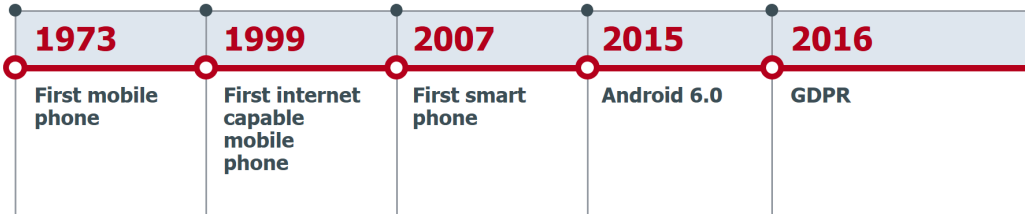


Figure 1.1: Mobile phones and privacy timeline

Overtime the Android operating system introduced multiple layers to protect user privacy. In the early versions, only applications from the official store, the Google Play Store, were available and at the very beginning, the store did not have any kind of regulations. Quickly, malicious agents realized the users did not care about the permission access information and flooded the application store with software to grab personal information in order to provide from this sensitive data.

With the growth of the store, in addition to the information on what kind of functionalities of the device are accessed, warnings were added before the download started, however, these warnings did not improve as most of the users did not understand the dangers related with each permission.

In addition to these updates to the store, more steps were taken to improve the user experience and by 2012, Google updated their developer policies and started cleaning up the offered applications [22]. It was at this time, that applications started to go through an automated review process in order to be approved to be offered in the store. Unfortunately, these efforts were not perfect and some malicious applications could get through. In 2015, this review process was changed from automated to manual in order to fight malicious, exploitative, inappropriate, and low-quality apps [31].

It was not the last time that Google updated their store as for each time one type of exploit was blocked another one would appear, making this a never-ending work. One example of this effort happened in 2018 when Google banned on applications that would mine cryptocurrencies in the background while draining resources from the user device. [9]

By Android 6.0 (Marshmallow), in 2015, it had evolved much since its release and many updates brought security features like Permission Manager[16] to make the users more conscious about what is being accessed by the applications. But this feature did not solve the complete problem since it lacks a method for displaying the intended purpose for each permission used.

Currently, Android has grown to be the most popular Mobile Operating System with 88% of the market share [35] and is one of the most attractive platforms for personal data collection.

A large number of permissions present in today's Android Operating System (OS) is hindering the users capability of comprehending what is happening and shared by the various applications in their smartphones. We propose the usage of introspection within the Android applications, via code injection using Aspect-Oriented Programming (AOP), to measure the data collection and notify the user or/and sink into a secure back-end.

1.1 Motivation

Today many applications are available for free on the Google Play Store and some of these gather money from advertisements or selling collected data. This data is collected by requiring extensive

lists of permissions that in the majority of the cases are not necessary for the purpose of the application [40].

With these problems in mind, an effort to regulate this kind of practices was started in 2016 by the states of the European Union and the European Economic Area by the form of the General Data Protection Regulation (GDPR) [30].

The article 1 of chapter 1 of GDPR [30] lays down rules relating to the protection of natural persons regarding to the processing of personal data and rules relating to the free movement of personal data. The GDPR also states various concerns on how the consent should be handled and the most relevant for our problem are:

- A request for consent should be clear, concise and not disrupt the use of service. - Recital 32 [30]
- If the processing is based on consent, the controller should be able to demonstrate that the data subject has consented this usage. Article - 7 [30]
- Any information that might be used to identify a person should be evaluated as a critical factor against its usage while taking into account the available technology and technological developments. The amount of time required and the costs should be parameters in this evaluation. - Recital 26 [30]

While Android has a Permission Model, it does not automatically make the applications GDPR compliant as it requires that the consent to process the data should be clear and concise.

1.2 Proposed Solution

In this section, we describe the objectives and features of the proposed solution to our problem. With the problem in mind, our solution is named after *Sobek*, an Egyptian deity considered to be a fierce protector that wards off the evil and defend the innocent.

1.2.1 Objectives

The main goal of this project is to enhance the user privacy while using their Android smartphone, as such we have the following objectives:

- **Literature review**: By reviewing the state of art we aim to identify the progress done to combat the problem and learn the pros and cons of each approach to propose a more efficient solution.
- **Architecture design**: The definition of architecture is fundamental, it helps to comprehend how the requirements of the solution relate to the implementation and the data flow between components should act.

- **Implementation**: The effort to implement the proposed solution should be able to combine the most recommended practices found in the research together with the planned architecture. This implementation should deliver a tool which offers improvements to our problem.
- **Security and privacy analysis**: In the end, we should look at the proposed solution and its implementation to analyze the security and privacy improvements made.
- **Deployment in a real environment**: Taking into consideration the most used application of the Play Store, conduct a study regarding the support, performance overhead and information leakage of the application.

1.2.2 Features

The main features of our project are as follows:

- **Easy migration**: The users should be able to use their settings independently on what device they are using.
- **High privacy level**: With this solution, we aim to provide an improvement for the smartphone's users on how their information is accessed and used.
- **High availability**: The solution should be able to be used anywhere, anytime, online or offline. While online, the servers should always be accessible.
- **Interoperability**: The connectivity between devices and applications remain unaffected as any changes do not alter the process of communication, only the sensitive information is modified.
- **Remote control**: The solution should be able to offer remote control capabilities as it can be a highly pursued feature for certain use-cases.
- **Profiling**: The creation and usage of user-defined profiles with fake sensitive data is crucial to deliver an intrusive free experience on some applications.

1.3 Contributions

This thesis aims to improve on the current state-of-the-art by making the following contributions:

1. **Literature review:** While reviewing similar implementations, we aim to identify the progress done to combat our problem described in Subsection 1.1 and learn the pros and cons of each approach to propose a more efficient solution.

2. **Related work classification:** In order to have a better idea on the state-of-art we categorize and score each of the found implementations. This classification has an emphasis on two main features, degree of usability and reliability to affect applications.

3. **Creation of a prototype for an Android security system:** We used the knowledge obtained from the literature review and related work classification to created Sobek, an security system that instruments Android applications in order to control the access of sensitive data to improve the confidentiality and privacy in the devices.

4. **No rooted solution:** The ease of usability heavily impacts the adoption rates from the users, as such our solution does not require a rooted device. Another concern with rooted devices is malware can gain unauthorized root privilege through exploiting an uninformed user heavily impacting the security of the data inside the device [39].

5. **User interface application:** With the usability in mind we created an application that helps the device user to control the behaviour of the instrumented code, either by allowing the original behaviour or modifying it.

6. **Improvements on the data confidentiality and privacy:** Our implementation allow the user to protect their contacts, SMS, location, and Wifi network address from applications by either faking the results or denying them.

7. **Taxonomy:** With this we to establish a relation from how applications are built and with the results obtained for our implementation.

1.4 Outline

This thesis is divided into four more chapters and are organized as it follows:

Background: Brief overview of some of technologies like Android, Aspect Oriented Programming, code obfuscation and cloud computing. This chapter ends with an review of other implementation as related work aiming to get a better insight on the subject.

Implementation: Details on the produced Android security system prototype. We describe process of creation of our multiple components, the issues, solutions and methodologies used.

Results: In this chapter contains types of tests conducted. The first tests detailed are against sample applications created specifically to evaluate the prototype. After we conduct the same tests on publicly available applications and provide an overview on the results obtained. Based on the findings we then propose an taxonomy for Android applications and evaluate the related work against our prototype. At the end we present an analysis of the performance impact on the smartphone and then we evaluate the performance and scalability of our back-end solution.

Conclusion: Evaluation of the work produced, listing limitations, possible future work and open research challenges.

Chapter 2

Background

In this chapter we explain some of concepts needed before approaching the problem. We start by explaining some relevant features and architecture of the Operating System (OS) of our devices followed by some programming ideologies. Further, we show some technologies that might be relevant to the creation of the final solution.

2.1 Android

Android is an open source, Linux-based software stack created for a wide array of devices and form factors [17]. This stack can be also referred to as a mobile operating system as its structure and functionalities resemble a traditional OS. This OS is fragmented into various layers (Figure 2.1):

The **System Apps layer** is where the core applications reside and are distributed with the OS and provide methods to utilize the most basic features of a smartphone like calls, texting, keyboard or web-browser. These apps are not any different from an application developed by a third-party other than being the default behaviour of the device.

The **Java API Framework** provides the developers with a set of tools to ease the development of applications by enabling the use of components, services and core functionalities specially developed to interact with the OS.

The **Native C/C++ Libraries** remain the fundamental for the system as they constitute the foundations of its components like the Android Runtime or the Hardware Abstraction Layer (HAL). These libraries are equally accessible to the application developers through the Java Application Programming Interface (API) Framework.

The **Android Runtime** is an abstraction that enables each application to run in a separated virtual machine for itself. This way each application has its own specific memory allocation and garbage collection making the execution smoother by optimizing these processes. Before Android 5.0 (Lolipop) this abstraction was made through the Dalvik Virtual Machine which was a similar method and the major difference was on the compilation of the app code going from just-in-time

to ahead-of-time reducing the start up time while increasing the installation time.

The **HAL** is made of multiple modules to communicate with the various hardware components. Each one of these modules is responsible for only one component and can be accessed by the Java API framework.

Finally, the **Linux Kernel** lays the foundation for the system, providing a proven, well-known hardware interface.

Based on this analysis, the most relevant layers for this project are the Java API Framework and the HAL as they are where the most applications rely upon to acquire the information that they request and where reside the most of the functionality to collect information about the surroundings of the device.

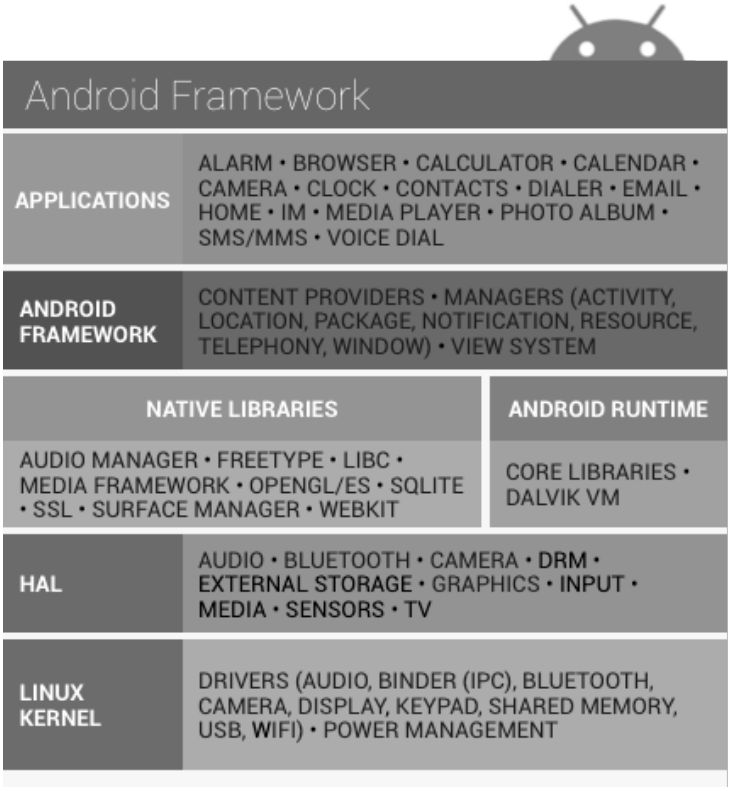


Figure 2.1: Android Platform Architecture[1]

2.1.1 Rooting

Rooting is a process of allowing Android device users of having privileged control over the system. With this, the user can have better customization of the themes, more control on the kernel level definitions, remove system apps, allow third-party apps to access or modify assets that other way would be impossible or even install a completely new firmware.

[1] https://source.android.com/security

Although what this process brings to the user experience, the most of the manufacturers do not support it as it leaves many options for inexperienced users to alter that can make their devices unusable either by software deficiency or hardware degradation. As a non-supported feature, this process can be hard for the people without technological background making its advantages out of reach for most of the population.

2.1.2 Permissions Model

Android 6.0 Marshmallow introduced a new permissions model that lets apps request permissions from the user at runtime, rather than prior to installation [19]. As Figure 2.2 shows this approach gives both the developer a method to explain the user the consequences of the decisions being made and the impact they have on their system.

This new model represents an upgrade since it offered the users the option to only allow the applications to access the requested data when the request is made but also to select which permissions were given, and has proven to have an impact in the user choice of applications [25].

However, this model does not offer an option to monitor what is done with the accessed data and after the permission is given only through the system settings it can be revoked, allowing the application to freely use the permission without alerting the user again. This model also does not automatically make the applications General Data Protection Regulation (GDPR) compliant since the permissions are to access the data or features and never to allow the information to be sent outside the device for further processing.

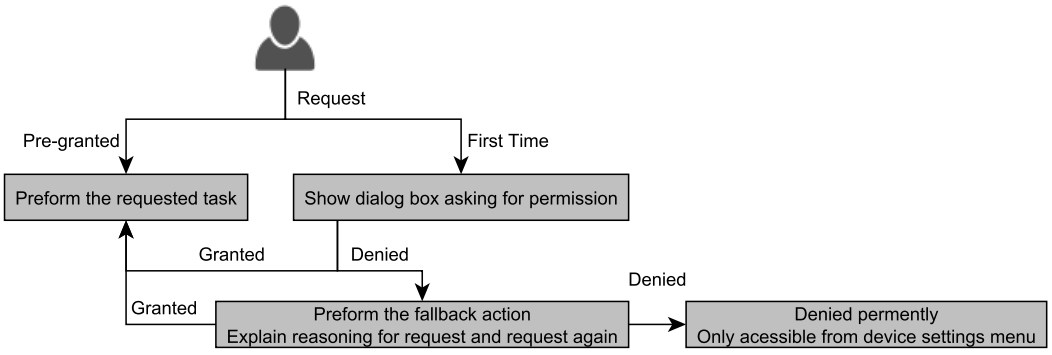


Figure 2.2: Runtime Permissions - User Flow

2.1.3 Project building

Since the early days of Android, Google has tried to make sure that development for the platform was easy to start by creating tools to deal with the complex build cycle required to create an application. One of these tools is the Android Gradle plugin that was created to extend the Gradle powerful build automation system with Android specific features.

However, to be able to do an analysis of any application, it is important to understand the whole build process that turns all the code and resources into one application that is able to be used by the end user. As shown in Figure 2.3, this process can be broken in two smaller processes, one that deals with all the code, and another to deal with resources such as images/videos and .xml files containing information about the application (AndroidManifest.xml) or static data.

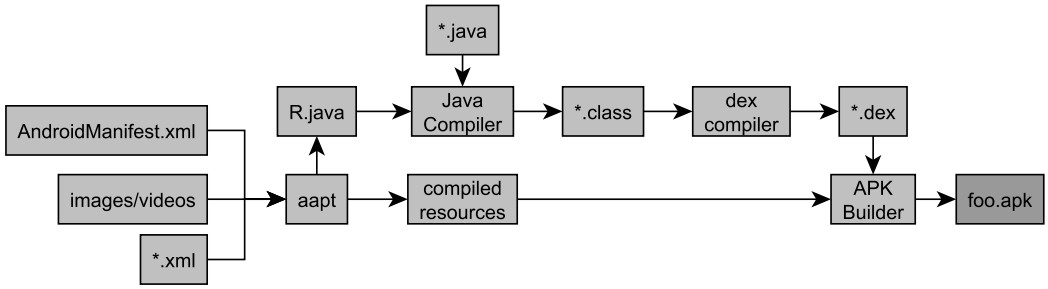


Figure 2.3: Standard Android project building process

2.2 Aspect Oriented Programming

The *introspection* is the ability to examine selected aspects of the structure, behaviour, and state of a system by the system itself [5]. This ability was further expanded and gave birth to a new architectural pattern, the *reflection*, that provides a mechanism for changing the structure and behaviour of software systems dynamically. It supports the modification of fundamental aspects like type structures and function call mechanisms [5].

These concepts represent the base of the Aspect-Oriented Programming (AOP) as it inherited their abilities and objectives. AOP is based on the idea that computer systems have better performance when each component is programmed for a specific concern and then tied together with the use of AOP into a coherent program [10].

To completely understand the AOP one must understand the concepts of *aspect*, *cross-cutting concerns*, *advice*, *join point* and *point-cut*. An *aspect* is a feature of a program that is used in many other parts of the program. *Cross-cutting* concerns are aspects of a program that affect other concerns. An *advice* is a class of functions which modify other functions when are triggered. *Join point* is a specification of when the aspect code should be executed. *Point-cut* is a group of join points.

One of the most popular and advanced implementations of AOP for Java is the AspectJ. It was created by the Eclipse Foundation. It first appeared in 2001 and is a project still in development getting frequent updates keeping up with the new concepts in the field by incorporating new features on each release. It is widely used nowadays in the Java development and is one of the few that is able to support Android applications.

2.3 Code Obfuscation

Code obfuscation is a process that tries to modify the compiled code in order to prevent the decompilation and further human analysis while maintaining all the original functionality [7]. This processes are mostly used to keep secret how the program works and prevent malicious agents to exploit vulnerabilities. Code obfuscation techniques usually are used when building releases of software together with code reduction techniques. In Figure 2.4 an example of this process can be seen and shows the impact of this process and the obstacles it presents against human comprehension or code matching techniques as all variables and functions have been renamed into single characters.

Original Code

```
public class test1 {
  private int term1;
  private int term2;
  private boolean areRelativelyPrime;

  public test1(int term1, int term2){
    this.term1 = term1;
    this.term2 = term2;
    areRelativelyPrime =
      areRelativelyPrime();
  }

  private static int
        gcd(int term1, int term2) {
    int remainder;

    remainder = term1 % term2;
    if (remainder == 0) {
      return term2;
    }
    else {
      return gcd(term2, remainder);
    }
  }


  private boolean
    areRelativelyPrime()
    {if (gcd(term1, term2) == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Obfuscated Code

```
public class a {
  private int a;
  private int b;
  private boolean c;

  public a(int a, int b) {
    this.a = a;
    this.b = b;
    c = c();
  }


  private static int b(int a, int b) {

    int c;

    c = a % b;
    if (c == 0) {
      return b;
    }
    else {
      return b(b, c);
    }
  }


  private boolean c() {

    if (b(a, b) == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Figure 2.4: Code obfuscation example[7]

2.4 Cloud computing

The definition of cloud-computing is constantly evolving, however we decided on a simplified version and describe it as any solution delivered as services over the Internet coupled with the all hardware and software used in the data centers [24]. This new paradigm introduces two concepts, the software that is able to be interact with the end-user usually is considered as Software as a Service (SaaS) and the hardware coupled with low-level software as Infrastructure as a Service (IaaS). The Figure 2.5 illustrates an simplified view of these concepts and how they are intertwined.

In the context of this thesis the proposed solution aims to be a SaaS that should be used on top of a IaaS solution such as Google Cloud Platform (GCP), Microsoft Azure or Amazon Web Services.

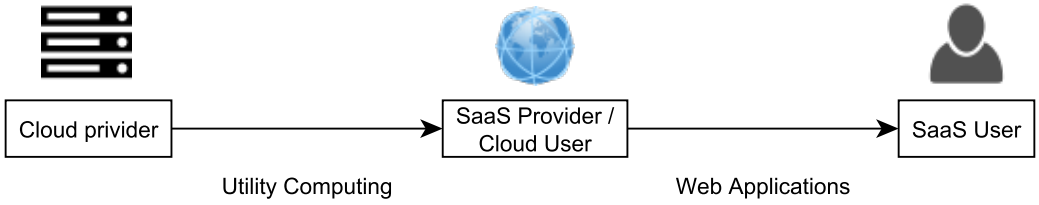


Figure 2.5: Users and providers of cloud computing

2.4.1 Micro-services architecture

The micro-service architecture is a design pattern for developing an application in small standalone modules [28]. This approach is the opposite of the traditional monolithic architecture and enables new methods of deployment, better dependency management and better problem tracking of the code base.

The improvements on the deployment start by being able to run as many as needed instances of a module behind a load balancer[2] meaning that we can better optimize the usage of our resources.

Another advantage is the dependency management as each independent module can be developed without concerns related to external requirements, like libraries or third-party software, as each have their dependencies bundled with the module.

Another improvement is that is easier to track down problems related with code since usually each module is responsible for a very specific task and finding the code responsible for the problem should be easier than going through the the whole code base of an monolithic approach.

[2] A load balancer is a component responsible for the distribution of network or application traffic to a cluster of services

2.4.2 Docker & Kubernetes

Docker is a software that performs vitalization on the operative system level. To do this, *Docker* grabs a container image and then proceeds to start a container which is an isolated environment with an application and its requirements.

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [18].

The combination of these two technologies enables highly scalable applications that are able to adapt themselves reacting to the demand. One of the most frequent uses is for web services with architectures similar to the one shown in Figure 2.6 where the demand is mostly unpredictable and the components can be run independently and have distinct needs in their scalability.

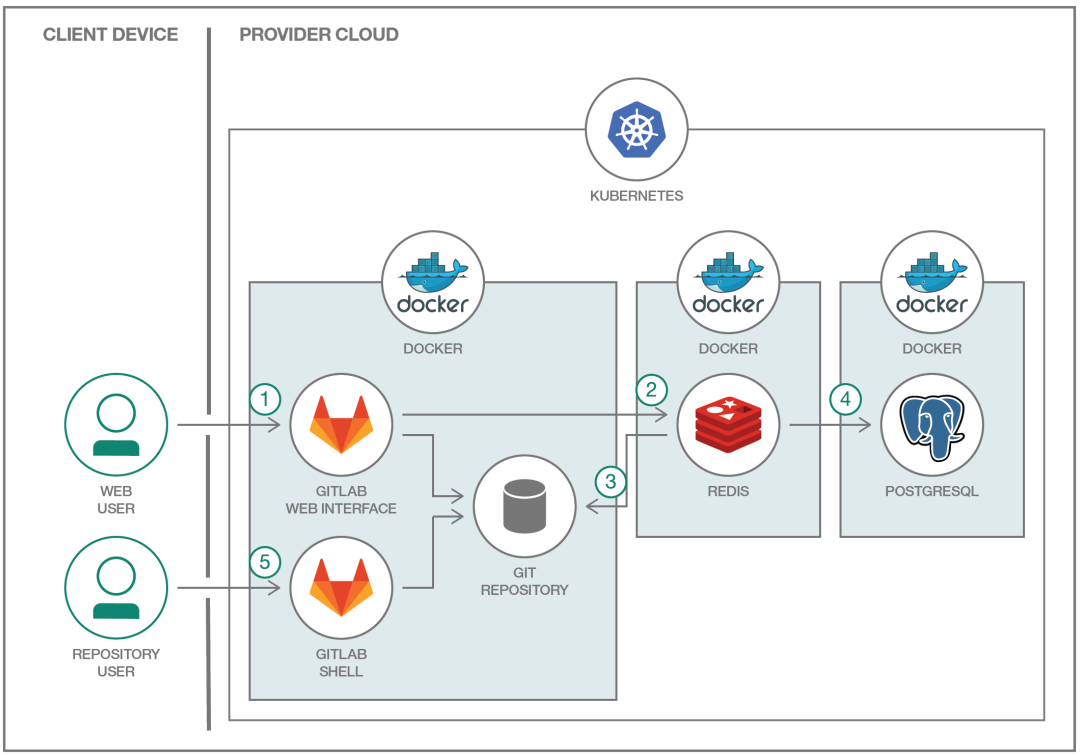


Figure 2.6: Docker + Kubernetes Architecture Example[3]

[3] https://medium.com/@AnimeshSingh_74972/microservices-polyglot-apps-websites-scalable-databases-hosted-repositories-bring-them-all-on-2f99eecb9039

2.5 Related Work

In this chapter, we present various implementations that tackle our problem. We start by reviewing similar applications aiming to get a glimpse of what and how this problem has been tackled and learn the pros and cons of each with the objective of introducing a more efficient approach. At the end of this review we compare all of these to measure the progress made in order to implement some features.

2.5.1 Research Methodology

In order to review similar applications, we defined a set of keywords according to features defined in Section 1.2.2. This set of keywords include the following:

- Android
- Instrumentation
- Aspect Oriented Programming
- Sensitive Information
- Weaving
- Security
- Personal Information
- Privacy Enhancement
- Permission Manager
- Monitoring

The search engine used was the Google Scholar coupled with regular Google Search Engine. With the results found in the search with these keywords were filtered by date, being only considered results from 2010 onwards. Additionally, some references showed up multiple times while reviewing but not in the search result and were consulted by an individual case in order to evaluate their relevance.

2.5.2 Similar applications

In this subsection we present the most prominent results of our research and provide a little insight on how they work.

TaintDroid
TaintDroid is a taint tracking system for Android designed to monitor how third-party applications handle the user sensitive data by categorizing the sources from where they can be accessed as *taint source* and the last step where they are used as *taint sink* [11].

This solution uses various kinds of approaches, variable-level to taint data while leaving the code untouched, message-level tracking to avoid Inter-Process Communication (IPC) overhead and extend the analysis to the system, method-level tracking that allows the propagation of taint with the return of the functions and lastly file-level tracking to ensure that the information retains

the taint. Each tracking level is attached to a layer of the Android architecture, message-level is placed between application code, variable-level on the virtual machine, method-level on the native system libraries and lastly the file-level tracking acts on network interfaces and secondary storage. The usage of these concepts in TaintDroid architecture can be seen in Figure 2.7 where can be seen the flow of data from the source up to the sink. Although the concepts applied in this system are important for addressing the problem the implementation lacks many common user-features like logging or even warnings.

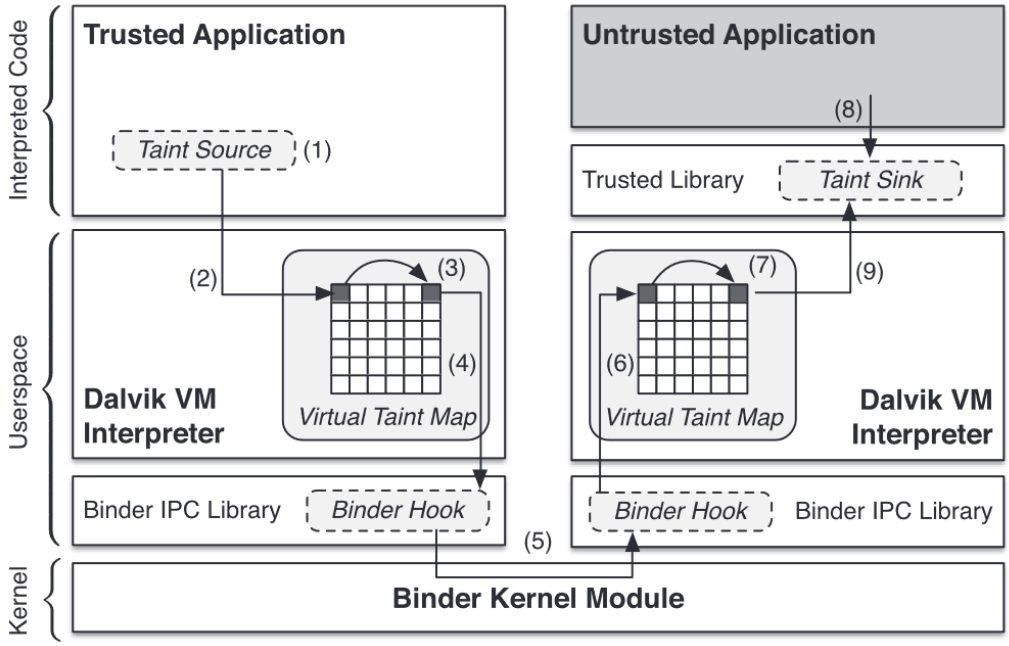


Figure 2.7: TaintDroid architecture within Android [11]

MockDroid

MockDroid is a modified version of the Android operating system which allows a user to 'mock' an application's access to a resource [4]. This version of Android includes a modified package manager which allows the creation of two kinds of permissions at the installation of a new application, one for regular permissions other for mocked permissions. These mocked permissions, are derivatives from the regular permissions and are used to allow the user to modify them while not altering the originals. The mocked data is stored within a new operating system user called *mock* which can be accessed through the mocked permissions and modified by the Mocker application that is a user-friendly interface. Some of biggest downsides of this approach are the root requirement and the complicated installation process while not providing features like logging.

PmP

Protect my Privacy is an application designed to infer the context around data accesses [6]. Its first appearance for Android appeared in 2016 and allowed users to control app-level permissions and lacked features to collect the stack traces of sensitive data and cloud syncing that were later released in an updated version in 2017.

The architecture of this project (Figure 2.8) shows the components of the system and specifies the interaction between the monitored application and the PmP. The PmP application gives a User Interface (UI) to allow the user to check and modify the current permissions and relay them to the PmP service, so it can save these changes to the Decisions Database. The Monitor service is responsible to have the decisions loaded and prepared to reply when a protected call is triggered and send the logging information to the server. The Anonymizer is where PmP generates credible new fake data.

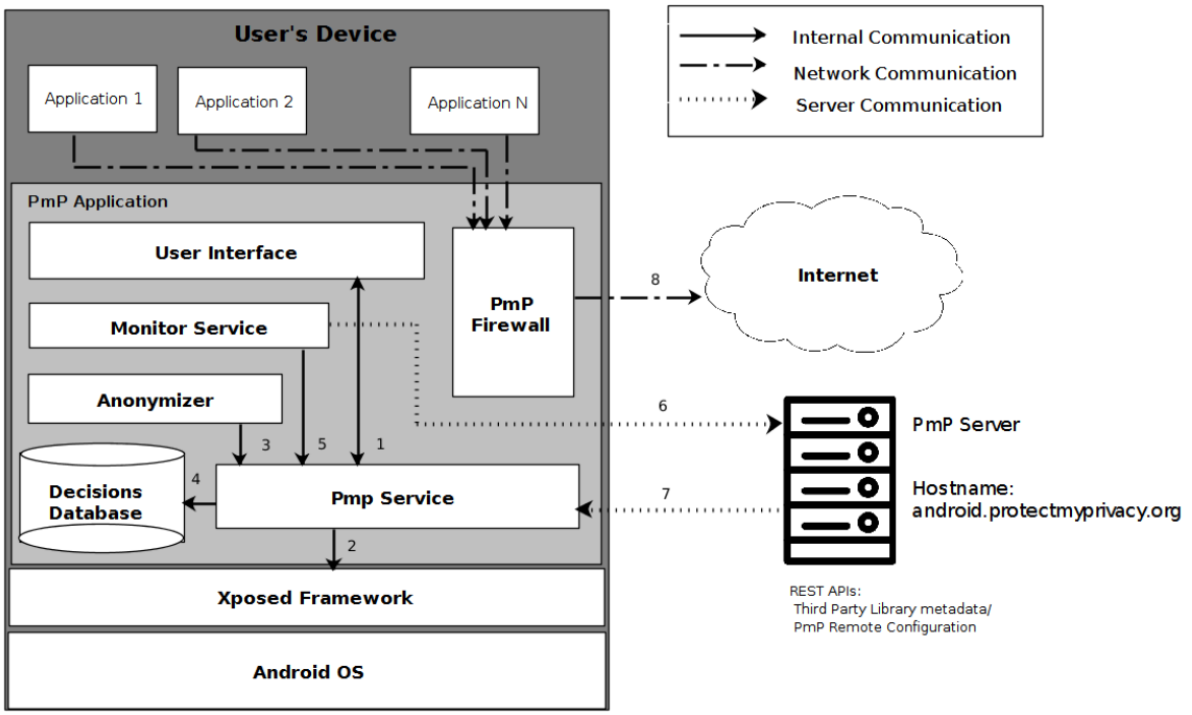


Figure 2.8: Architecture of PmP [6]

While these components are built upon the Xposed[4] to modify the calls to where the sensible data is stored, the PmP Firewall is used to complement the lack of interceptions for mobile data / Wifi connections as it interacts with the default iptables[5] to force the user decisions on these connections. The PmP Server is a storage mechanism for the decisions made in the application and relies on a MySQL database to store the information and Secure Socket Layer (SSL) to secure the connections. This information is stored with a one-way hash of the device id as the only identifier of the user and does not store any personal information.

Unfortunately, this project is close-source however with the article is possible to have some insight on how to achieve similar results. One major drawback of this application is that the device must be rooted to have Xposed installed to work.

[4]Xposed is a framework that allows users to easily apply add-ons to the ROM.
[5]iptables is a Linux utility to manage Linux Firewall

RefineDroid + Dr. Android + Mr. Hide

These applications form an approach where the one application goes through RefineDroid to be inspected and create a set of fine-grained permissions that are used by Dr. Android to retrofit the standard Android permissions by replacing them [23]. Afterwards, these fine-grained permissions are managed by Mr. Hide that acts as a drop-in replacement for sensitive Android API's. The flow of how these components are integrated can be seen in Figure 2.9. The usage of Mr.Hide service showed us a solution for how the behaviour can be decoupled from the instrumented code had provide an simple method of altering it without going through the whole process again.

This approach allows the user to protect their information, but does not provide remote control or logging capabilities. Unfortunately, we were unable to find information if this project had cloud-based instrumentation and as close source project we could not test ourselves.

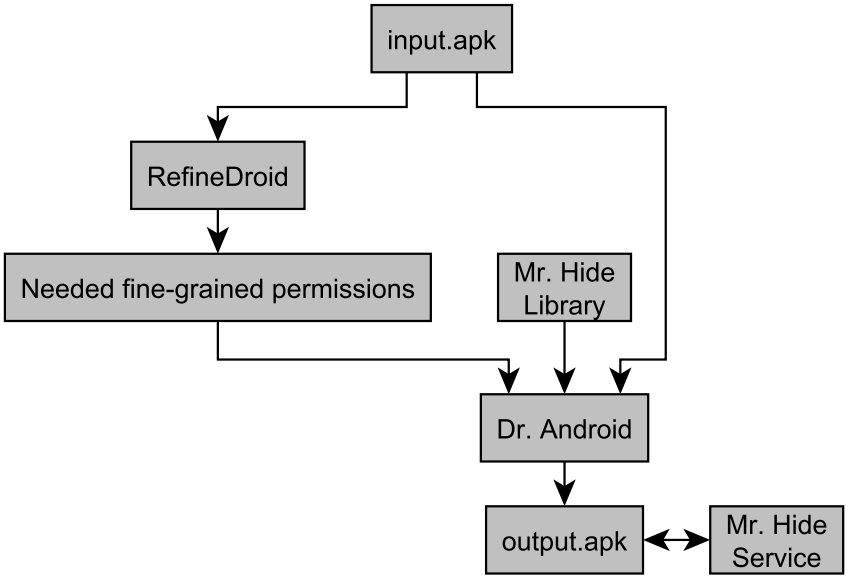


Figure 2.9: RefineDroid + Dr. Android + Mr. Hide architecture

RV-Android

This project tries to enable run time verification on Android devices by expanding RV-Monitor technology which is a project by the same developers that allows the code to be monitored by a set of rules defined in either aspects or javaMOP [8].

The build process in Figure 2.10 refers to the various components / tools and the inputs and outputs of each. Through the usage of javamop, Monitor Oriented Programming Files can be converted into AspectJ Files able to be used with AspectJ Compiler to instrument any application.

It allows the applications to be instrumented in the device or in a cloud server but does not confer the power to give simple user control over how their information is accessed or to provide modified information.

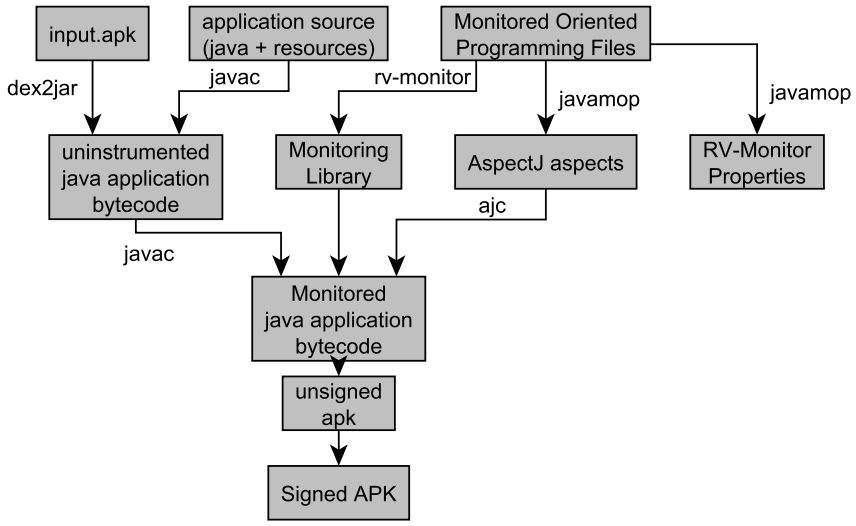


Figure 2.10: RV-Android Build Process

Weave Droid

Weave Droid is an Android application that makes AOP on Android devices possible and user-friendly [12].

This application provides an easy way to use AOP however it does not have user-oriented functionalities. The article of its release describes many relevant concerns and solutions, however, the project seems to be halted as the Android Package (APK) and source code are not public at the time of writing.

Figure 2.11 represents its architecture and provides an insight on how an online and offline solution can be achieved. Through the usage of embedded weaving coupled with a local aspect repository and a local application repository the offline usage is guaranteed. For the devices with fewer capabilities an online solution is present with the use of an external server (Weaving Cloud) for the cloud weaving with an external aspect repository and external application repository. All these features are delivered through WeaveDroid application.

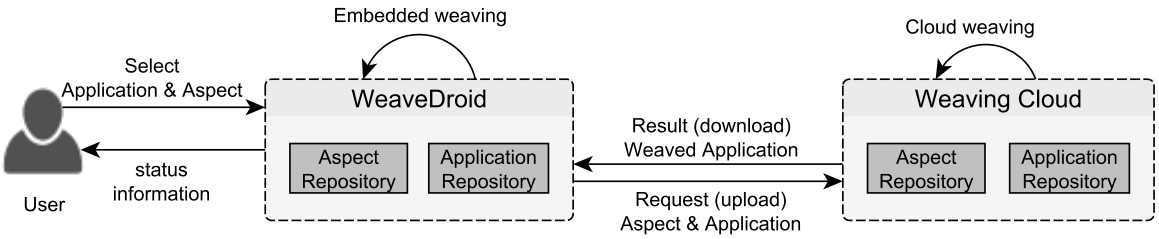


Figure 2.11: Weave Droid context

Adrenaline-RV

Similar to WeaveDroid this project aims to provide instrumentation on Android, however, it does not use AOP but DiSL [26] and relies on an external server to instrument the code [37] as displayed in Figure 2.12. The Adrenaline-RV does not provide user-oriented features and as the latest version is not compatible with the most updated versions of Android. It also is discontinued since November 2017.

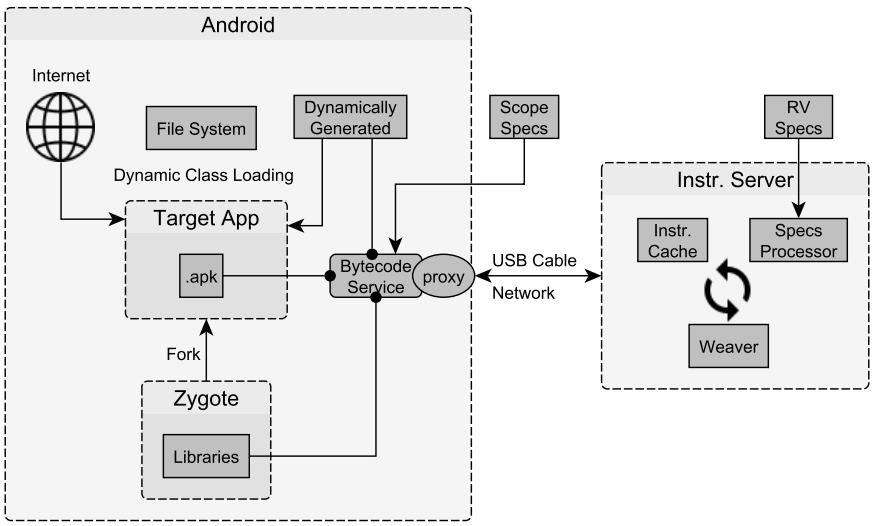


Figure 2.12: Architecture of Adrenaline-RV

SRT-AppGuard

This close-source project tackles the problem by rewriting the application to add monitoring capabilities in itself that are able to be read by the application. It is discontinued since March 2018 and only supports Android 2.3 to Android 4.4 [2].

In Figure 2.13 is displayed a concept of how monitoring of one application can be implemented with the usage of one service coupled with a visual monitoring application.

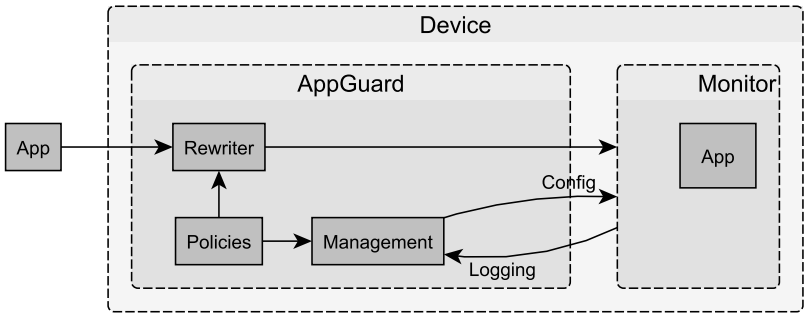


Figure 2.13: Schematics of SRT-AppGuard

2.5.3 Related work comparison

To be able to compare the various reviewed application we first decided a set of features that we think that are a must to tack the problem described in Section 1.1. These features are:

- **Easy migration**: The users should be able to use their settings independently on what device they are using.
- **High privacy level**: With this solution, we aim to provide an improvement for the users on how their information is accessed and used.
- **High availability**: The solution should be able to be used anywhere, anytime, online or offline. While online, the servers should always be accessible.
- **Interoperability**: The connectivity between devices and applications remain unaffected as any changes do not alter the process of communication however only the sensitive information is modified.
- **Remote control**: The solution should be able to offer remote control capabilities as it can be a highly pursued feature for certain use-cases.
- **Profiling**: The creation and usage of user-defined profiles with fake sensitive data is crucial to deliver an intrusive free experience on some applications.

However this features were too abstract in order to implement, as such we narrowed them into a more concise objectives that could be tackled in a direct way. In Table 2.1 depicts the relations between this objectives with the proposed objectives.

- **No root required**: This objective aims to ease the migration between devices as some devices may be rooted and others not and thus we should focus on what is granted.
- **User defined policies**: In order to provide high privacy levels and profiling we consider that the user should be able to create their own policies for each behaviour supported by the instrumentation as it enables better control on the personal information.
- **User friendly interface**: An user interface should be provided as it enables the most common smartphone users to use our system without requiring previous knowledge on the subject. This objective should ease the user when migrating their preferences and creating new profiles.
- **Allow to fake data**: We require the that the solutions should be able to provide fake data to the other applications as it enhances the user privacy and is crucial for the implementation of profilling as each profile should have its own set of fake data.
- **Realistic fake location**: Being able to provide realistic fake locations to other applications allows the solution to reliably provide this type of information without detection or enable

the user to not only fake their position at a singular time and position. This may enable the usage of the solutions in applications that need to calculate the traveled distance without revealing the true location, increasing the privacy level for the user.

- **Cloud-based policies**: These enable the solutions to be easy to migrate between devices and enhance the interopability between system. They also provide high availability since as long as the device can be online their settings can be used. Another enhancement is that this allows for remote control of the instrumentation behaviour through other devices than itself.
- **Remote logs storage**: By storing the logs remotely we diminish the burden in the device storage as these can be cleaned regularly freeing space and enable another methods for consulting outside of the device.
- **Cloud-based instrumentation**: The instrumentation process should be usable off-device to provide better compatibility and ensure that all devices can obtain instrumented applications regardless of their specifications.
- **Sensitive data access logging**: Logging the accesses done by applications to the sensitive data allows for a better understanding on how and why this is requested improving the user knowledge of the applications behaviour.

	Easy migration	High privacy level	High availability	Interoperability	Remote control	Profiling
No root required	x		x	x		
User defined policies		x				x
User friendly interface	x					x
Allow to fake data		x				x
Realistic Fake Location		x				x
Cloud-based policies	x		x	x	x	
Remote Logs Storage	x			x	x	
Cloud-based instrumentation	x		x			
Sensitive data access logging	x					

Table 2.1: Comparison Features Criteria

Additionally, we choose to add the type of the project, open or close source, as open-source projects can be used to build upon or give a glimpse on how to build a new solution based on previous iterations. The criteria of working in up to date Android was used because it is

important to give a solution appropriate for every device available and also help to prove the resilience of the solution against updates. In conclusion, the table 2.2 summarizes the features present on all the reviewed implementations and we could aggregate the implementations by their objective:

- **Tools for development**: WeaveDroid & Adrenaline-RV - These do not offer any kind of UI and are not targeted for regular smartphone users.
- **Research prototypes**: TaintDroid & MockDroid & RefineDroid + Dr. Android + Mr. Hide - Created as proof of concept these projects do not enhance the end user experience with their devices as they do not provide functionalities for the day-to-day usage.
- **Commercial applications**: PmP & RV-Android & SRT-AppGuard - Functionalities that aim to improve the end user experience are present coupled with methods for the user to interact with the implementation.

Additionally we found out that the none of these did not offer any sort of logging or cloud-based policies, and very few gave the user the option to fake data.

Feature \ Solution	TaintDroid	MockDroid	PmP	RefineDroid + Dr. Android + Mr. Hide	Rv Android	WeaveDroid	Adrenaline-Rv	SRT-AppGuard	Sobek
No root required	○	○	○	●	●	●	○	●	●
User defined policies	○	●	●	?	●	○	●	●	●
Open-source	○	●	○	○	●	○	●	○	●
User friendly interface	●	●	●	?	●	○	○	●	●
Allows to fake data	○	●	●	●	○	○	○	○	●
Realistic Fake Location	○	○	○	○	○	○	○	○	●
Logging	○	○	●	○	○	○	○	●	●
Cloud-based Policies	○	○	○	○	○	○	○	○	●
Remote logs storage	○	○	●	○	○	○	○	○	●
Cloud-based instrumentation	○	○	○	?	●	●	●	○	●
Works on up to date android (API-28)	?	○	●	?	○	○	●	○	●

●- feature present; ○- feature absent; ? - feature unclear;

Table 2.2: Similar implementations comparison

In the end we the most promising for users turned out to be PmP, however, it had one major flaw for the problem we are aiming as it requires root. Most of them gave us ideas on how we can achieve our goal either through the tools used, processes needed or architecture of the whole system. An additional insight on how these implementations behave on the current smartphones can be found in Subsection 4.9 where we test and scale them based of their performances.

2.6 Android Decompilers & Compilers

From our research done in Section 2.5 we learned that to achieve a solution we are required to decompile the applications in order to obtain code that can be instrumented and then recompiled. In this section we describe various tools that are able to decompile fully developed application and compilers for Android.

2.6.1 Decompilers

apktool is a tool for reverse engineering 3rd party, closed, binary Android apps. It can disassemble resources and repackage all back to .APK format. It is build in Java and uses either *AAPT* or *AAPT2* (Android Asset Packaging Tool). By disassembling an APK it is possible to read/write the XML files inside it and reach the .dex files that can be used by other tools to inspect and modify the source code. After all modifications this tool can repackage the APK back to its original state with the result being an unsigned APK.

Dex2jar is a tool for converting Android's .dex files (Dalvik bytecode) into .jar files (Java bytecode), enabling the access to the .class files within the application. This tool is also distributed with others such as *jar2dex* that provides a way to reconvert the files into .apk, *d2j-asm-verify* to verify the integrity and that the files are valid APK files and *d2j-apk-sign* to sign the .apk with a new pair of keys. The development of this tool started by 2010 and entered in maintenance mode by 2015 which means it does not support many of the most recent features of Java language. Although the results are acceptable most of the times, when used on APK that utilize newer Java features, or have edge cases it may fail or produce incorrect results, making these unusable for further instrumentation and recompilation.

Similarly to *Dex2jar*, *enjarify* is another tool to translate Dalvik bytecode into Java bytecode and was built to allow Java analysis tools to analyze Android applications. This tool is written in Python, however the author has a Rust and Go languages prototypes. It is recommended to use *PyPy* as interpreter instead of *CPython* to ensure the best performance possible. At time of writing only version 35 dex files are supported which means that Java 8 bytecode features are not supported and does not translate optional metadata such as sourcefile attributes, line numbers, and annotations. These restrictions mean that like the previous tool, it is possible to obtain acceptable results in many cases however it can produce incorrect results. The restriction on annotations translation is a huge drawback because some of the most popular libraries for

Android development, like Retrofit2 and Picasso, rely on these to work properly meaning that this tool is unusable on APK with these libraries.

2.6.2 Compilers

DX is the original Android compiler bundled with Android Software Development Kit (SDK) that is responsible to generate Dalvik bytecode from .class files [38]. Because it is the default Android compiler, it must retain backwards compatibility and to meet this requirement, Java 7 onwards is not directly supported, however, any feature from these versions need to get desugared[6] into compatible code.

Proguard is a Java tool that can be used to shrink, optimize, obfuscate, and pre-verify Java code [33]. This tool is normally used when building an APK in release mode to shrinking and optimize the APK before releasing to the public access to ensure that the user gets the best performance while preventing unnecessary usage of device storage. The obfuscation feature can be used to add an layer of protection against reverse-engineering by renaming classes, field and methods to random short names as described in Section 2.3. A general overview of the process of using both *dx* with *Proguard* is represented in the Figure 2.14.

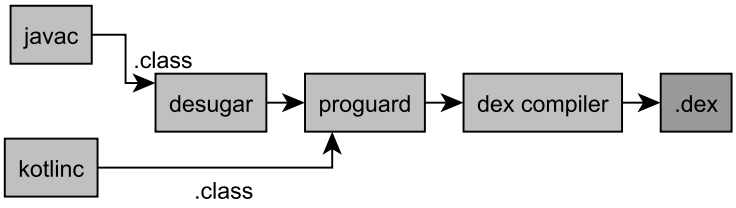


Figure 2.14: DX + proguard compilation

d8 is a tool used to compile Java bytecode into Dex bytecode and was created with the purpose of reducing the long build times which were needed to provide the previous compilers with more recent Java features [13]. The reduced build time is achieved by moving the process of desugaring into the compilation process as can be seen in Figure 2.15.

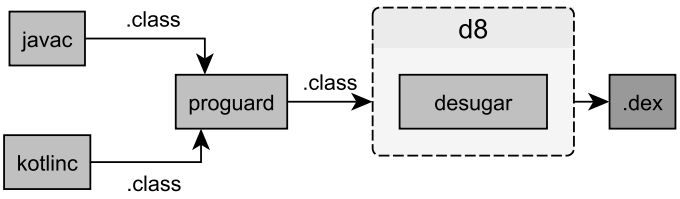


Figure 2.15: D8 compilation

[6]Desugar - transformation of source code into a nore syntactically rigorous form

r8 is an upgraded version of *D8* as it serves the same purpose of compiling and desugaring Java bytecode while being able to provide *Proguard* features like code reduction, optimization and obfuscation, making it a all-in-one tool reducing the build times even further [15].

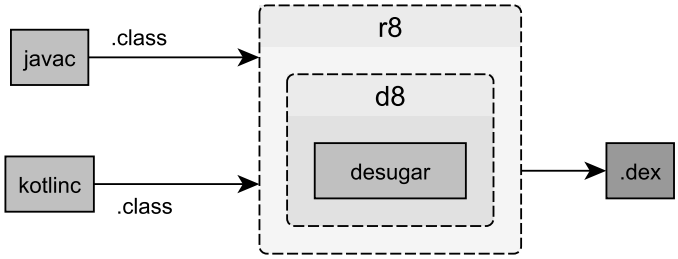


Figure 2.16: R8 compilation

2.6.3 zipalign & apksigner

zipalign is an archive alignment tool that is responsible for one last optimization before distribution. This optimization is achieved by ensuring that all uncompressed data starts with a particular alignment relative to the start of the file. This procedure reduces the RAM consumption of an application when running.

In order to distribute one application, the APK must be signed so that the authorship can be verified and thus protecting the users against fake or modified replicas. This is where *apksigner* comes allowing the to sign. While signing, the signature is also verified to work in every specified version of Android.

Chapter 3

Implementation

In this chapter we describe the multiple components of our Android security system.

The system is composed by an Android user application, that is responsible for controlling the instrumentation behaviour, a backend service that enables synchronization of preferences across multiple devices along with providing access to the Sobek Instrumentation Tool that receives normal applications and instruments them.

In Figure 3.1 we display an overview of these components and how they are connected. Each one of these components are explained in detail in this chapter, being split into the Sobek Instrumentation Tool (Section 3.1) where the code modifications are applied, Sobek Manager (Section 3.2) the application that controls the behaviour of instrumented applications and the backend (Section 3.3) that aggregates all server side components for serving instrumented applications, synchronization of settings and logs.

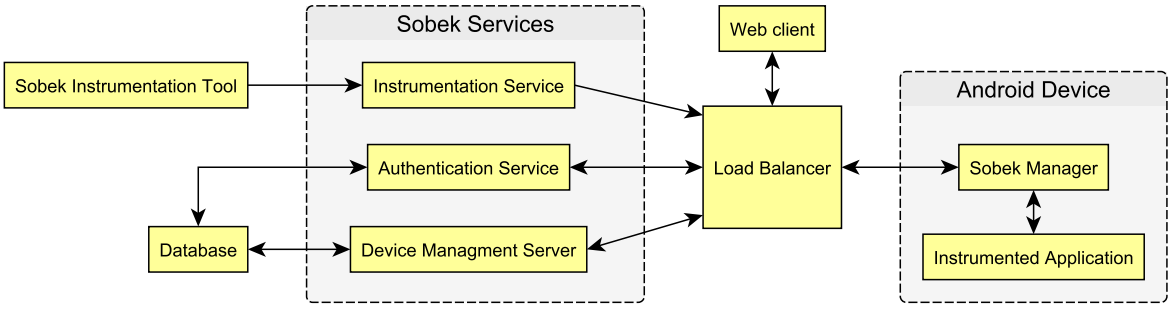


Figure 3.1: Sobek System Overview

3.1 Sobek Instrumentation Tool

In order to instrument applications, we built a tool based on the ones reference previously. This tool was built as a shell-script written in Bash[1] and has the following requirements:

- *javac* - Included in Java Software Development Kit (SDK)
- Android SDK
- zip and unzip can be installed with the help of a package manager such as apt/yum/rpm
- sed - stream editor for filtering and transforming text usually included in the Linux distributions.
- AspectJ (subsec 2.2)
- apktool, enjarify and dex2jar (subsec 2.6.1)
- r8 (subsec 2.6.2)
- JAVA_HOME environment variable pointing to Java Development Kit (JDK) folder
- ANDROID_HOME environment variable pointing to Android SDK folder

In order to utilize his tool, a few parameters must be provided in the execution, with these being, the original .apk file path and then the path to the folder where the aspects are located. It is also able to receive an additional parameter to specify the decompiler to be used, *enjarify* (default) or *dex2jar* and another parameter to trigger the cleanup of the temporary and output folders used in process. Additionally the key-store password and alias must be setup inside the script in the variables, KEYSTORE_PASSWORD and KEYSTORE_ALIAS respectively.

The Figure 3.2 depicts how one instrumented application can be achieved from its original state. This flow can be separated into three distinct phases:

Resource extraction and code decompilation (1) where *Apktool* is used to extract both resources and compiled code which is fed into *dex2jar* or *enjarify* for decompilation;

Injecting and instrumentation of code (2) with the use of *javac* together with *ajc* in order to modify the input application based on the aspects provided;

Recompilation and repackaging (3) with the use of *r8* followed by *Apktool*, *zipalign* and *apksigner* to repackage all the resources back into an .apk. The specific tool for resource modification refers to an additional step we added using *sed* in order to change the google_maps_key, found in the .xml files, that is used by some applications that use Google maps library to display a map.

[1] Bash is a Unix shell and command language

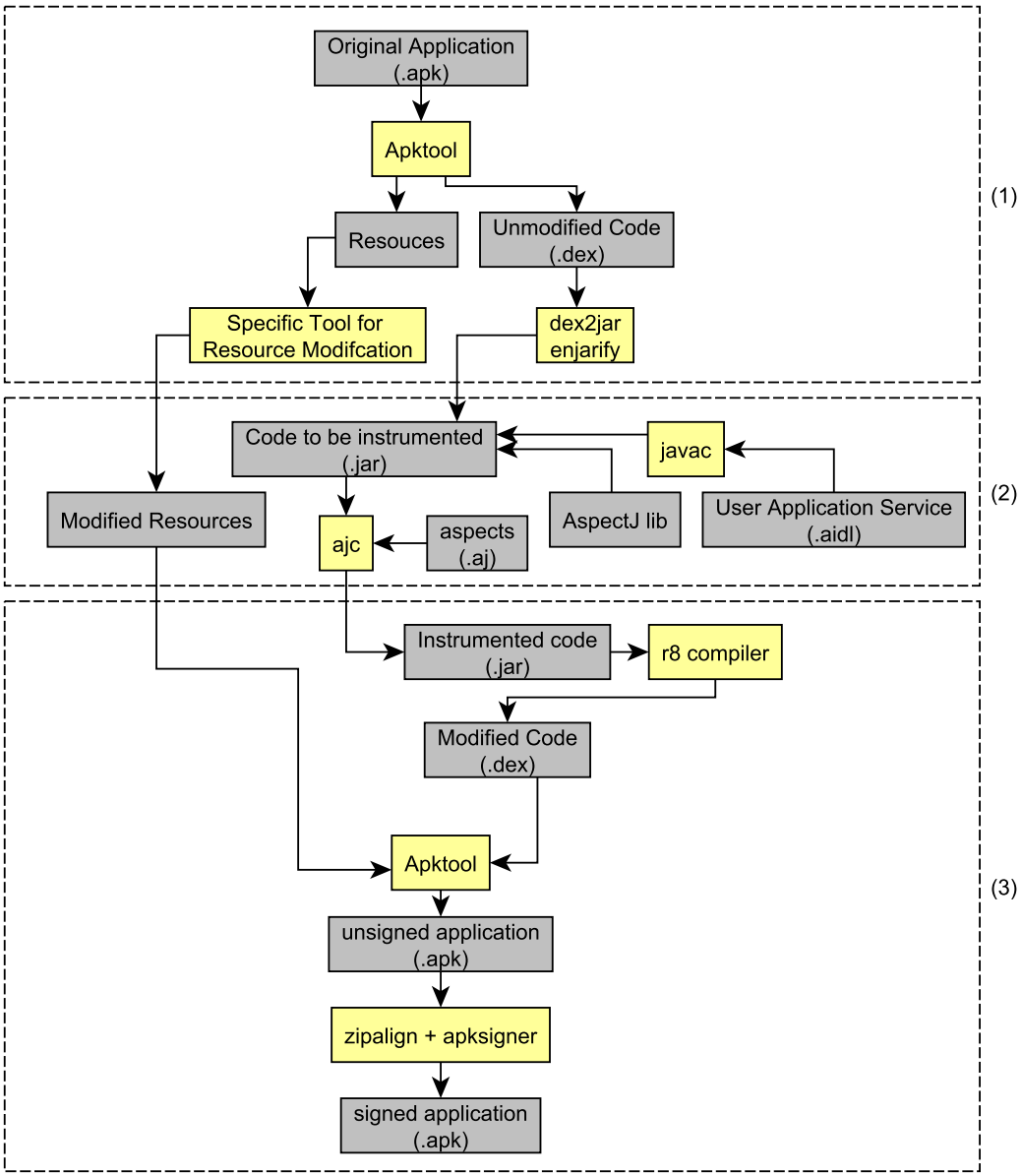


Figure 3.2: Sobek Instrumentation Tool work-flow

We started developing this tool based in the previously obtained knowledge from the related work review described in Subsection 2.5. As such we used *dex2jar* decompiler and *dx* Android compiler. However these tools revealed several problems while instrumenting applications and being unable to recompile due to missing classing classes errors or unknown functions.

In order to understand these errors, we inspected the decompiled code and compared them with issues found in *dex2jar* Github page and found that these were happening due to the lack of updates of the tool in order to support newer Android features. Some of these issues, recommended to try out another decompilation tool, *enjarify*.

With *enjarify*, we were able to make some progress and get code that was able to be compiled,

however we then stumbled upon new errors but this time these were during runtime. Must of these were about missing code related with Android SDK and thus we narrowed that the problem should be related with the re-compilation process.

As such we decided to use *d8* and *r8* in order to be able to utilize newer and more feature rich compilers to tackle the problem. *d8* did not reveal any progress against these errors as this compiler did not have code reduction, optimizations and desugaring incorporated. With the problem being the missing functions we used *r8* compiler with a custom rule-set in order to force the compiler to maintain the missing functions from Android SDK.

We started this rule-set with some of the most basic rules (Listing 3.1). Some of the most important ones in this rule set are the "keepatributes" and "keep public class" that force the compiler to keep attributes such as annotations, exceptions, innerclasses, etc and keep all the classes that extend some of the most used Android SDK classes such as Activity, Application or Service.

```
-dontusemixedcaseclassnames
-dontskipnonpubliclibraryclasses
-dontskipnonpubliclibraryclassmembers
-dontpreverify
-verbose
-allowaccessmodification
-keepattributes *Annotation*
-renamesourcefileattribute SourceFile
-keepattributes SourceFile,LineNumberTable
-repackageclasses ''
-keepattributes Exceptions,InnerClasses,Signature,Deprecated,SourceFile,
    LineNumberTable,*Annotation*,EnclosingMethod
-keep public class * extends java.lang.Exception
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class com.android.vending.licensing.ILicensingService
-dontnote com.android.vending.licensing.ILicensingService
-keep, includedescriptorclasses class
    in.uncod.android.bypass.Document { *; }
-keep, includedescriptorclasses class
    in.uncod.android.bypass.Element { *; }
```

Listing 3.1: Basic *r8* rules for Android

With further usage of these techniques we managed to have more reliable results, and kept adding new rules based on new finding of libraries used on Android applications.

An example of these custom rules we used can be seen in Listing 3.2. These rules are provided by Square, the company behind the creation of Retrofit2, in their Github page[2].

```
# Retrofit does reflection on generic parameters. InnerClasses is
    required to use Signature and EnclosingMethod is required to use
    InnerClasses.
-keepattributes Signature, InnerClasses, EnclosingMethod
# Retrofit does reflection on method and parameter annotations.
-keepattributes RuntimeVisibleAnnotations,
    RuntimeVisibleParameterAnnotations
# Retain service method parameters when optimizing.
-keepclassmembers,allowshrinking,allowobfuscation interface * {
    @retrofit2.http.* <methods>;
}
# Ignore annotation used for build tooling.
-dontwarn org.codehaus.mojo.animal_sniffer.IgnoreJRERequirement
# Ignore JSR 305 annotations for embedding nullability information.
-dontwarn javax.annotation.**
# Guarded by a NoClassDefFoundError try/catch and only used when on the
    classpath.
-dontwarn kotlin.Unit
# Top-level functions that can only be used by Kotlin.
-dontwarn retrofit2.KotlinExtensions
-dontwarn retrofit2.KotlinExtensions$*
# With R8 full mode, it sees no subtypes of Retrofit interfaces since
    they are created with a Proxy and replaces all potential values
    with null. Explicitly keeping the interfaces prevents this.
-if interface * { @retrofit2.http.* <methods>; }
-keep,allowobfuscation interface <1>
```

Listing 3.2: Retrofit2 r8 rules for Android

In order to surpass the issues described previously we decided that our tool, should support both *dex2jar* and *enjarify* usage as each application may have different behaviours due to each tool restrictions explained in Subsection 2.6.1 and issues.

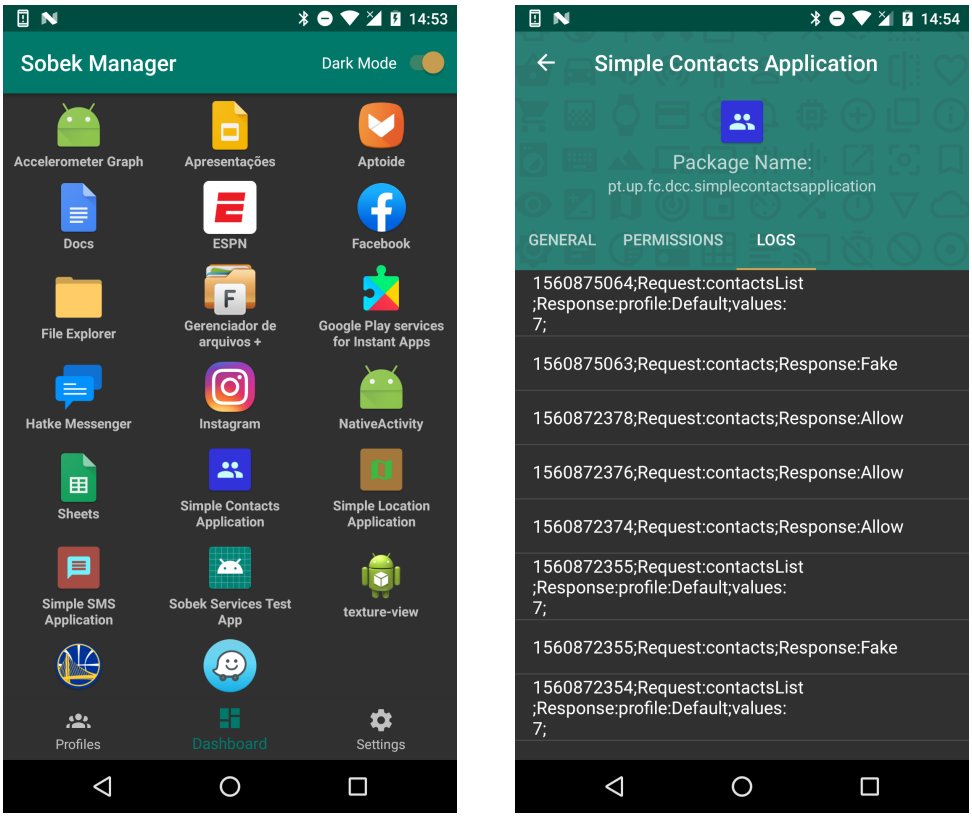
For .dex compilation, we choose *r8*, because of the all-in-one package it offers and being able to utilize all the improvements on the code reductions/optimization. With this, we are able to increase the optimization of some applications since their original state may not have utilized such methods.

[2] https://github.com/square/retrofit

3.2 User application

In order to accomplish the objective of having a user interface to define the behaviour of the instrumented code, we created the Sobek Manager, an application that is able to list every non-system application and allows to connect to an instrumentation server to instrument these applications, define per application behaviours and check the instrumentation actions logs.

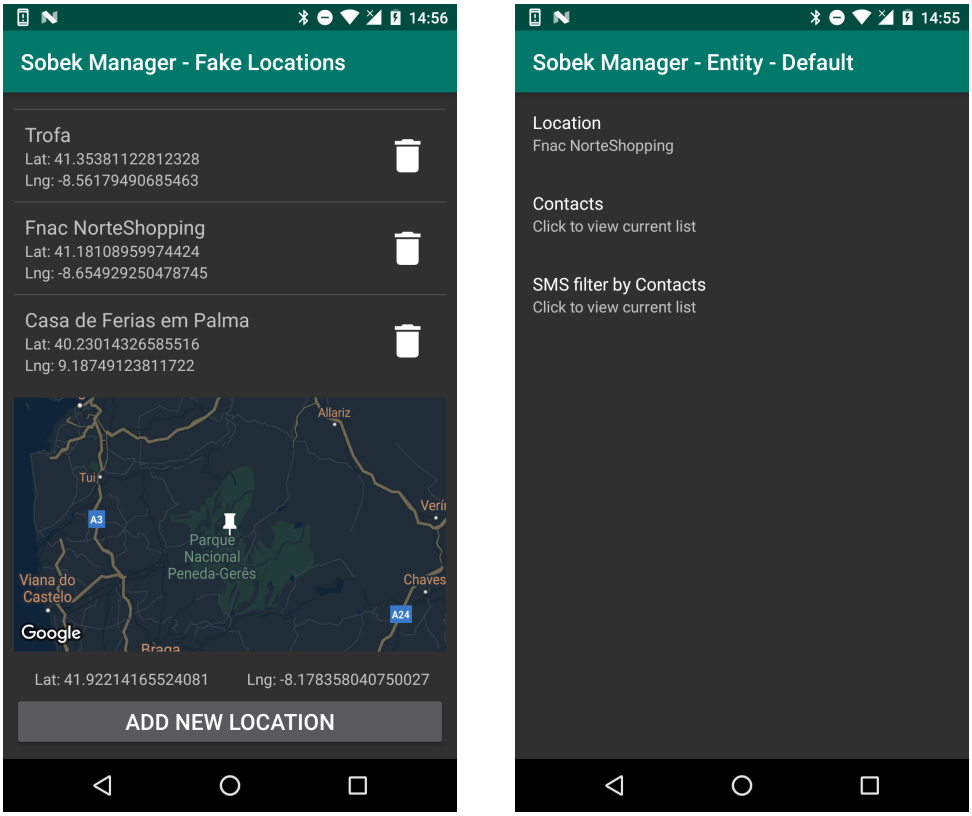
The Figure 3.3(a) shows the listing of all non-system applications and Figure 3.3(b) shows an example of instrumented actions logs in a sample application.



(a) Sobek dashboard (b) Sobek logs example

Figure 3.3: Sobek Manager

To accomplish our objective of being able to fake the device position we created an activity where a user can manage their fake locations. It is possible to add new locations with an identifier string and remove them when no longer needed. To add new locations the user have an fully functional map from Google GMS library. The Sobek Manager offers profiling for each setting (Figure 3.4(a)) available meaning that we can create fake personas to reliably lie to specific applications with fake locations (Figure 3.4(b)).



(a) Fake location settings (b) Profile specific settings

Figure 3.4: Sobek Manager Settings

3.2.1 Preferences and Logging

To save user preferences, we decided to use the default SharedPreferences interface of the Android SDK which allows us to access and modify preference data. This interface stores this information as files in a private folder for each application. As we have two distinct types of settings, we decided on various diferent files, the default SharedPreferences, *fakeLocations*, *entities* and *applicationsPreferences*.

As SharedPreferences work in a key-value pairs we decided to use Comma-separated values (CSV) style keys and values as it helps to have simple rules when creating new pairs for new applications,locations or profiles.

The default SharedPreferences aggregates Sobek Manager general settings.

fakeLocations saves the user defined fake locations with the key being the name and a CSV style for the coordinates, *latitude;longitude*, as value.

entities holds information related to each user defined profile.

applicationsPreferences stores the behaviour for the instrumentation on that application along with the profile to be used that holds the fake information.

Additionally each individual application has a CSV file that holds all the information about the interactions of the application with Sobek. In this file each line represents an event.

3.2.2 Inter-Process Communication (IPC) Service

We decided that instrumented applications should be able to be controlled as such we decided to utilize Android Services in order to provide IPC. An overall idea how this IPC mechanism works is displayed in Figure 3.5.

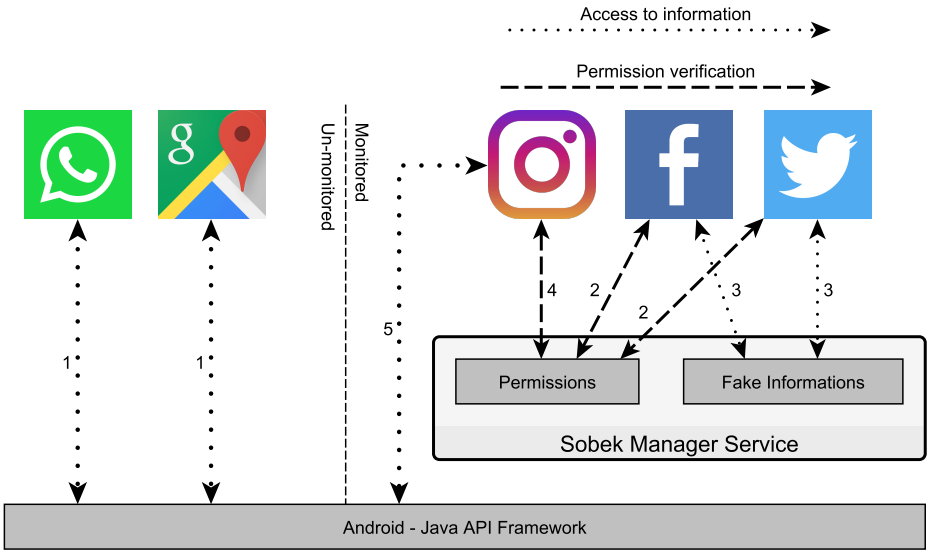


Figure 3.5: Applications Interactions with Sobek

On the left side of Figure 3.5 we choose to represent two of the most popular applications that requires access to device information to work while un-monitored. One of these is the WhatsApp that requires access to device's contacts to help the user connect with his friends. We also choose Google Maps as one of the most used applications for accessing maps and getting directions, and this one requires access to the device location. In both of these cases, the applications are used as delivered by the Google Play Store. After the user allows the applications to access the needed data they request it directly to the Android's Java API Framework for this information (1).

On the right side of Figure 3.5 are represented three applications being monitored by Sobek. In the first case, we used Instagram as an example of one application with location requirements where the user allows the access to the data both in the Android and in Sobek. In the Instagram application request for information, there is a new step where it is done the verification of accesses against the Sobek Permissions Service and is selected what action should be taken next (4). As the permission is granted by the Sobek Permission Service, the application is redirected to request the Java API Framework for the location (5).

The Facebook and Twitter applications represent applications that require many permissions, for the Facebook example we will review the case of faking contacts. The Twitter application

exemplifies an example where Sobek will fake the user location. Although the requirements are different, their interaction with Sobek is the same. First, they need to verify their permissions on Sobek Permissions Service (2). Afterwards, with the access to real data denied, the applications are instructed to request data from the Sobek Fake Content Provider instead of connecting with the Java API Framework (3).

In order to provide the previously explained solution we needed to create a class that extends Service class from the Android SDK and implements the functions defined in the AIDL file (Listing 3.3). Each one of these functions are accessed by the instrumentation code in order to query Sobek Manager preferences and save each interaction in a file as described in the previous Subsection 3.2.1.

```
interface ISobekAidlInterface {
    String accessType(String applicationPackage, String data);
    String fakeLocation(String applicationPackage);
    String allowedContactIds(String applicationPackage);
    String allowedSmSAddress(String applicationPackage);
    String fakeMacAddress(String applicationPackage);
}
```

Listing 3.3: Sobek User Application Service AIDL Definition

By default Android Interface Definition Language (AIDL), supports: All primitive types in the Java programming language, String, CharSequence, List and Map. In order to simplify our prototype we decided to use String for every type of values being transmitted between the processes and we expect them to be formatted as demonstrated in the Table 3.1.

Variable	Expected format	Meaning
accessType	^(Allow|Deny|Fake|FakeMovement)$	Permission type
fakeLocation	^(\-?\d+(\.\d+)?),\s*(\-?\d+(\.\d+)?);(\-?\d+(\.\d+)?),\s*(\-?\d+(\.\d+)?)$	latitude;longitude
allowedContactIds	^(\d{1,19})(;\d{1,19})*$	List of long integers separated by ;
allowedSmSAddress	^((\(?\+?[0-9]*\)?)?[0-9_\- \(\)])(;((\(?\+?[0-9]*\)?)?[0-9_\- \(\)]))*$	List of phone numbers separated by ;
fakeMacAddress	^[a-fA-F0-9:]{17}|[a-fA-F0-9]{12}$	MAC address
applicationPackage	^([A-Za-z]{1}[A-Za-z\d_]*\.)+[A-Za-z][A-Za-z\d_]*$	Application package name
data	^(location|contacts|smsFilter|wifiMacAddress)$	Type of data being accessed

Table 3.1: ISobekAidlInterface expected string formats

3.2.3 Synchronization with Backend

Android's Worker class permits to schedule the execution of tasks without compromissing the normal behaviour of the device. As such we created two classes *LogSyncWorker* and *PreferenceSyncWorker* that are responsible to synchronize both the logs and the preferences with our back-end through the usage of gRPC.

To enable the usage of gRPC we created a protocol buffer definition and used *protoc* (Section 3.3.1) to create stubs for all the functions there defined. These stubs were then used as base for the fully fledged implementation.

Additionally we made sure that the workers could have its schedules being periodic or one time, as the user may not want this feature to work automatically.

3.3 Backend

With multiple of our objectives being cloud-based, we decided on a micro-services architecture where each service is responsible for a very specific task as described in Subsection 2.4.1. The overall architecture is shown in Figure 3.6.

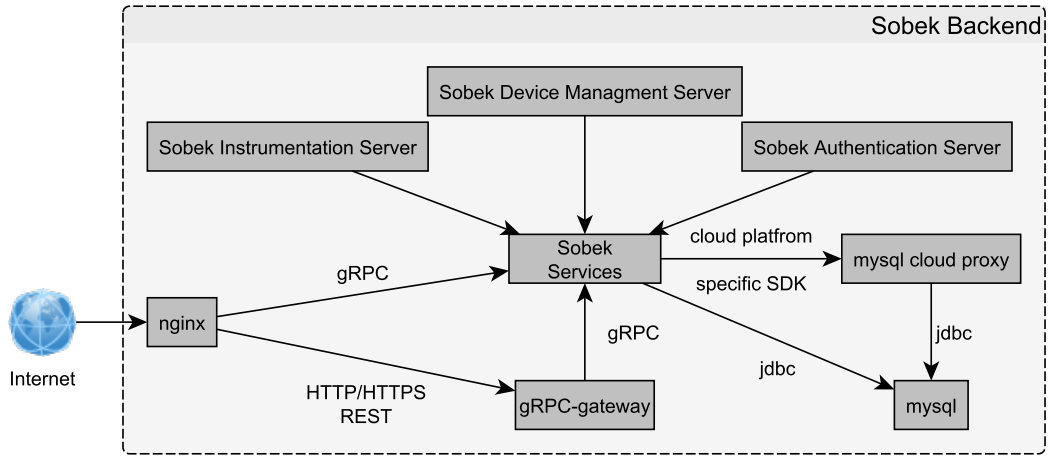


Figure 3.6: Sobek Backend

3.3.1 Tools & Frameworks

gRPC is an open source general purpose Remote Procedure Call (RPC) mechanism developed by Google [3]. gRPC uses protobuf definitions in order to establish the communication protocol, and is able to be used both in servers and clients regardless of the used language (Figure 3.7).

Protoc is a compiler that converts the protocol definition into .proto files that can be used to generate protobuf runtimes for multiple languages.

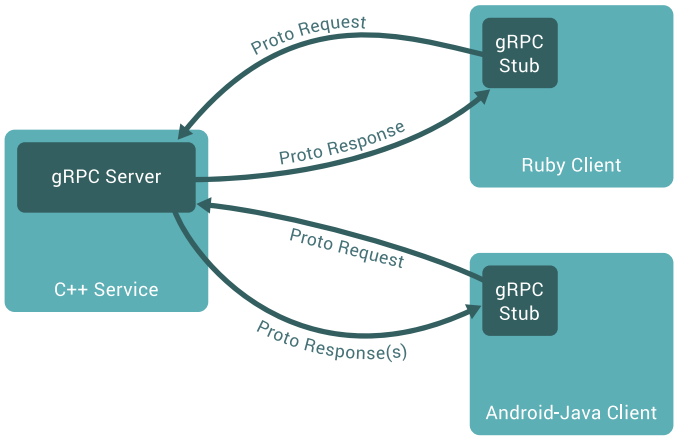


Figure 3.7: gRPC Usage Example[3]

gRPC-gateway is plugin for protoc that generates a reverse-proxy server able to translate Representational State Transfer (REST) Hyper Text Transfer Protocol (HTTP) Application Programming Interface (API) into gRPC. By using this plugin we can provide an REST API while reusing the gRPC services easing the implementation of an external front-end. An example of this usage is represented in Figure 3.8 where the gRPC service that provides the function "example.ProfileService.Update" is exposed through an RESTful API in "PUT /v1/user/123/profile" with the usage of gRPC-gateway. This plugin requires that the gRPC protocol definitions have the *google.api.http* annotations attached in order to define the paths for the requests.

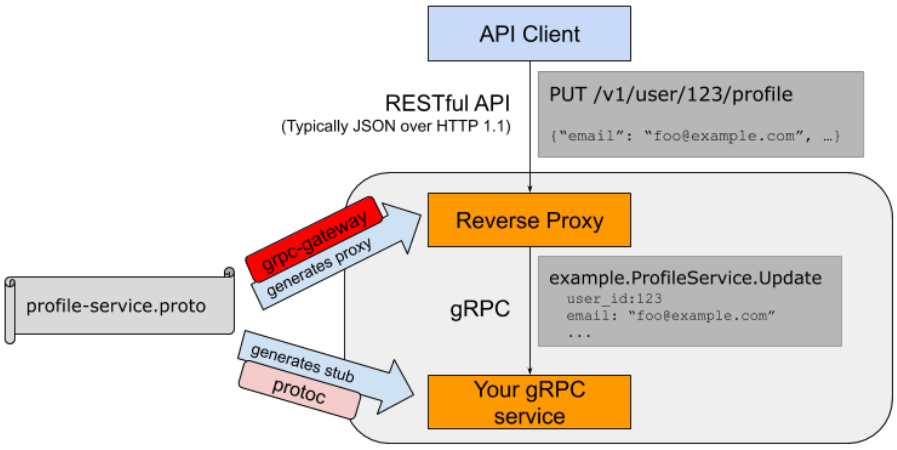


Figure 3.8: gRPC-gateway functionalities overview [20]

The *OpenAPI* is an open-source framework for defining and creating RESTful APIs. The API specifications can be written in JavaScript Object Notation (JSON) or YAML Ain't Markup Language (YAML) and can be used to create stubs of clients and servers reducing the conflicts between the components and the development time. One tool capable of creating these stubs is *Swagger-Codegen* that is developed by the original creators of the *OpenAPI*. In the early versions, the *OpenAPI* was called *Swagger*.

[3] https://grpc.io/docs/guides/

3.3.2 Database Design

The design of the database was though based on the different informations required to be saved.

Each device should be able to have its various preferences saved, as such we created *devices* table in order to hold the unique devices along with time-stamps for synchronization. Further we created table *locations*, responsible to hold fake locations, *event_logs* , saving all the events caught by the instrumentation, and *device_application_settings* in order to save each the instrumentation behaviour preferences for each instrumented application.

In order to provide each user authentication and functionalities to control multiple devices, we created table *users* that hold regular login and password information coupled with *users_devices* that aggregates each device, through *androidId*, to an user through *user_id*.

Lastly we created an last table *apps* that hold information about previously instrumented application in order to reduce the instrumentation process executions.

The Figure 3.9 displays an overall structure of the final database used in this implementation that achieves all of the previous statements.

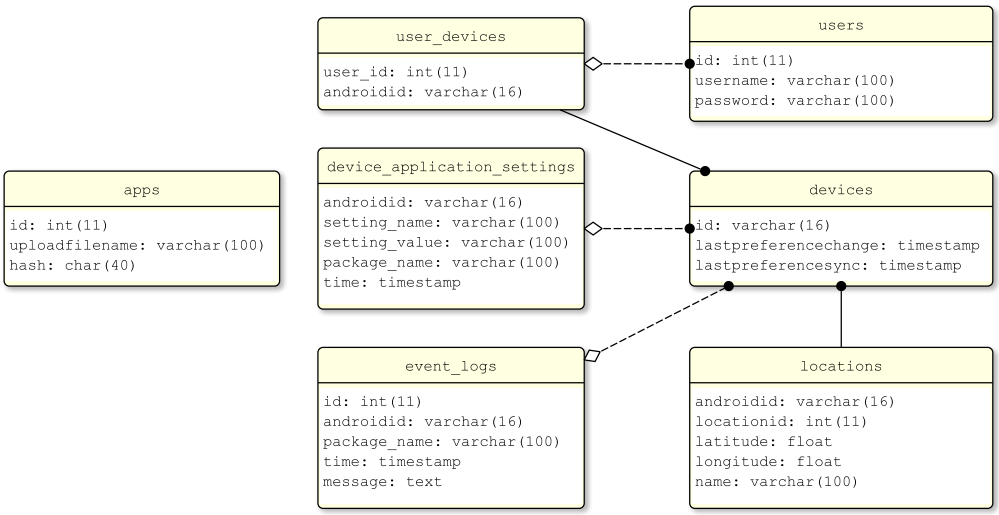


Figure 3.9: Sobek Backend Database diagram

3.3.3 Sobek Services

In order to provide our security system with functionalities like, out-of-device instrumentation and synchronization we created services according to our chosen micro-services architecture.

These implementations were developed in as a single Java program that has the ability to transform itself into the multiple services from the startup parameters. With all of these being available from a single program we aimed to ease prototyping and testing while in a future version these should be independent programs.

Instrumentation Service

The instrumentation service is responsible for the execution of the Sobek Instrumentation Tool (Section 3.1) on behalf of the devices. This service only has two functions, one for upload and another download (Listing 3.4).

The upload function is responsible for the transmission of the original application in .apk format from a smartphone to the backend and it is done by a streaming chunks of the file while returning the status of the operation.

The download function enables a smartphone to retrieve an instrumented application in .apk format from the backend and similarly to the upload function it is also a stream of chunks. After recieving an .apk this services starts by calculating the hash of the file and checks if there is an already instrumented file with this hash. This functionality allows the service to prevent the usage of resources in vain as we already have an instrumented version ready to be served to the requesting client.

```
message Chunk {
    string name = 1;
    bytes content = 2;
     int64  offset = 3;
}
enum UploadStatusCode {
    Unknown = 0;
    Ok = 1;
    Failed = 2;
}
message UploadStatus {
    string Message = 1;
    UploadStatusCode Code = 2;
}
message DownloadReq {
    string id = 1;
}
service InstrumentationService {
    rpc upload(stream Chunk) returns (UploadStatus) {
        option (google.api.http) = {
            post: "/instrument"
            body: "*"
        };
    }
    rpc download(DownloadReq) returns (stream Chunk) {
        option (google.api.http) = {
            get: "/instrument/{id}"
        };
    }
}
```

Listing 3.4: Protobuf Instrumentation Service Definition

Authentication Service

As this implementation is a prototype we decided that this service should only provide a

simple authentication that is responsible for login and register on the backend. This service implementation (Listing 3.5) only has the minimal requirements for the rest of functionalities to work.

```
message OperationResult {
    bool done = 1;
    string message = 2;
}
message LoginCredentials {
    string username = 1;
    string password = 2;
}
service AuthService {
    rpc login(LoginCredentials) returns (OperationResult) {
        option (google.api.http) = {
            post: "/login"
            body: "*"
        };
    }
    rpc register(LoginCredentials) returns (OperationResult) {
        option (google.api.http) = {
            post: "/register"
            body: "*"
        };
    }
}
```

Listing 3.5: Protobuf Authentication Service Definition

Device Management Service

This service aggregates the functionalities of synchronization of preferences and logs from and into the user application (Section 3.2). Similarly to the previously described services, this also uses an gRPC definition (Listing B.1) for the communication protocol.

3.3.4 Docker Images

With *Docker* we created images that could be ran anywhere *Docker* is installed without having to concern with the hardware and the software present on the machine. Since we decided on a micro-services architecture, we could build as many images as necessary for each component, meaning that each image could be specifically tailored for each case optimizing the size and performance. For the Sobek Backend we decided on 4 images as being the core of our implementation:

- Instrumentation
- Authentication / Device Management
- Nginx Server
- gRPC Gateway

We start building from the official Ubuntu image for Docker and adding both 32-bit and 64-bit support in order to increase the comparability of the image. Afterwards we add the needed

repositories to the package manager and installing all the different requirements of our tools and server code. One special case is the Android SDK that is not installed through the package manager and is downloaded from the official source. The next step is to add *Enjarify* and *dex2jar* from their git repositories. Then we start adding our Sobek related missing pieces such as the Sobek Instrumentation Tool, Sobek Manager Service AIDL file, aspects and the backend code.

The Sobek Authentication / Device Manager image is just a simple image built from the official open-jdk Docker image with our backend code.

Nginx server image is the standard official Nginx Docker image with modified configuration files that meet our needs.

Lastly the grpc-gateway is built from golang official Docker image with added grpc specific libraries and our generated entrypoint.

3.3.5 Kubernetes Setup

In order to deploy the previous Docker images (Subsection 3.3.4) on our chosen Cloud service provider, we decided to use Kubernetes to create a computing cluster. In Kubernetes, we have workloads and services.

Each one of these workloads holds a micro-service composed by the service itself and some optional helper services.

Additionally to the workloads, Kubernetes have services that are responsible for establishing endpoints so each micro-services can connect with others as the architecture requires. These services can also be used as load balancers. In our deployment we decided on the usage of one entry-point for each services together with two load balancers, one for the nginx server and another for instrumentation micro-services, as both of these services can be accessed from the Internet.

Through the usage of Autoscaler, we were able to scale the number of pods in a replication controller [1]. Based on the profile of our tool execution obtained in the Subsection 4.11.3 we choose the default behaviour (CPU utilization) for this scaling.

3.3.6 REST API

Through the usage of gRPC-gateway and OpenAPI (described in Subsection 3.3.1) we are able to provide an REST API to interact with our services in the backend. As we did not developed an web-based client to interact with this, we used Postman[4] that could load the OpenAPI definitions created by *protoc*. These API is saved as a .json file and can also be used in tools such as Swagger Editor[5], in order to visualize how these calls can be made and their parameters.

[4]Postman is an application for API development and testing (https://www.getpostman.com/)
[5]https://github.com/swagger-api/swagger-editor

Chapter 4

Results

To test our implementation, we decided on two distinct types of applications, sample apps that we had access to the source code and publicly available apps from the Google Play Store. We required that these applications requested permissions for location, contacts, Short Message Service (SMS) or Internet.

The idea behind testing on applications that we have source code is to be able to debug and understand how the instrumentation behaving. On the other hand we also used applications from Google Play Store to measure the impact of the instrumentation in a real world scenario.

4.1 Simple Location Application

This sample application[1] was created in order to track the behaviour when instrumenting code around the most popular methods to find the device position:

- getLastKnownLocation(String provider)
- getLastLocation()
- requestLocationUpdates(...)

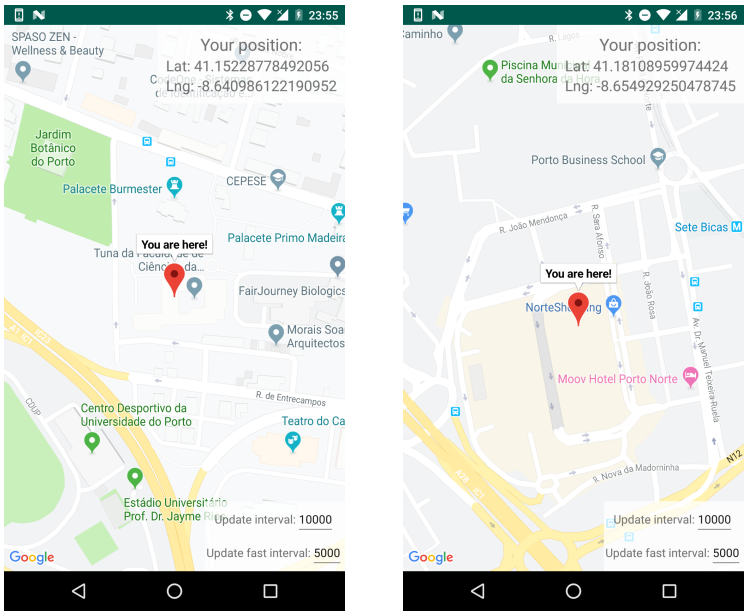
getLastKnownLocation and getLastLocation() are two functions that return an object of Location class, where the latitude and longitude of a device can be retrieved by getLatitude() and getLongitude(). In contrast the method requestLocationUpdates(...) is used to setup triggers when the device updates its location. This setup usually use LocationRequest with LocationCallback to provide the developers access tho the device position. From the callback LocationResult is returned which contains an array of objects of class Location.

In our sample application we choose to use the method of requestLocationUpdates(...) as it is the advised method in the Android Software Development Kit (SDK) documentation and

[1] https://github.com/Evilong/SimpleLocationApplication

also because it should be the most generic, meaning that what if the instrumentation works on this case, it should also work on the others [14]. This simple application was developed based on previous statements and as shown in Figure 4.1(a) it can show the user its position on map from Google Play Services.

The instrumentation on this application was successful by using aspects with pointcuts on methods getLatitude() and getLongitude() of class Location and both decompilation tools worked on this case. With the instrumentation result we had an application that we could change the behaviour when it tried to get the device position (Figure 4.1(b)).



(a) Normal Position (b) Fake Position

Figure 4.1: Simple Location Application Screenshots

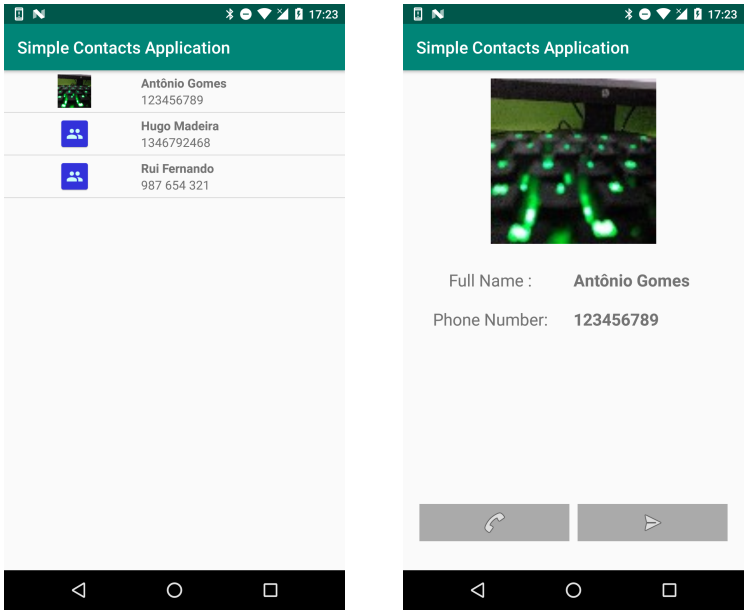
4.2 Simple Contacts Application and Simple SMS Application

Similarly to the Simple Location Application, we made an sample application[2] to test instrumentation on the access to the contacts stored in the device. In Android, the contacts are usually stored through the usage of content providers. This content providers are used to encapsulate the data have defined contracts where the various methods of accessing are established together with the permissions required. For the contacts an default content provider is available in Android. The data in this provider can be accessed by querying it through the ContactsContract. The result of this query encapsulates can be the entire information of a contact stored in the device or a few selected fields, like name, email or phone number.

In order to create this sample application, we used two activities, and we query the whole contacts content provider for its information, displaying a list of available contacts (Figure 4.2(a)).

[2] https://github.com/Evilong/SimpleContactsApplication

When clicking on a user, a new activity is created that uses the previously obtained information (Figure 4.2(b)). A simplification of the life-cycle of this application can be seen in the Figure 4.4.



(a) Contact List (b) Contact Information

Figure 4.2: Simple Contacts Application Screenshots

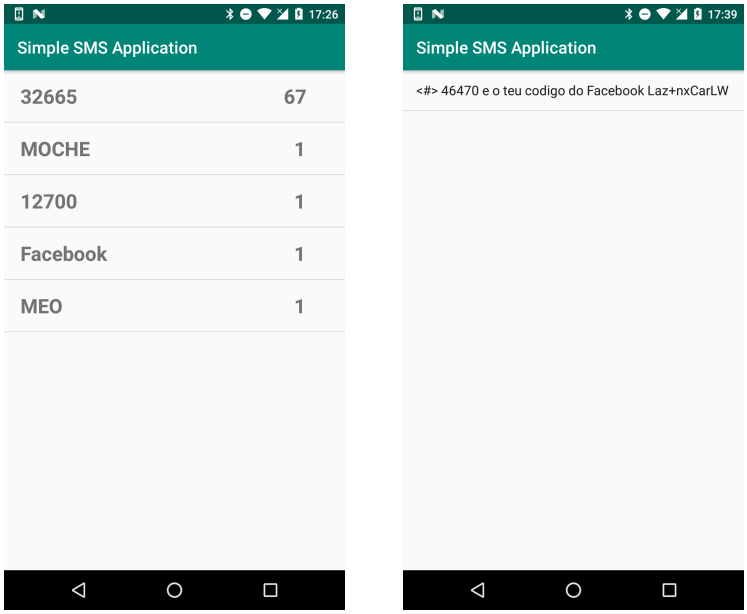
One source of personal data can be found in the SMS stored in the device and as such these applications can be used to retrieve this information. With this threat identified, we created the Simple SMS Application[3] to test the applicability of instrumentation in these applications.

The most common method to retrieve this messages, is similar to the previous example as it uses ContentResolver with Cursor to query the database identified by the URI of *content://sms/inbox*. Our sample application emulates this information retrieval by querying all the information and then grouping by the identifiable contact. The Figure 4.3 illustrates this behaviour.

The instrumentation on these applications was made by creating pointcuts on .query(..) function of Cursor class. These pointcuts and using around advices, we checked for the arguments passed into the function, and if it was being used to access the contacts information we could redefine the filters based on the configurations made in the Sobek Manager. This instrumentation modifies the original life-cycle (Figue 4.4) of the application and the modified version can be represented as in Figure 4.5.

The instrumentation was successful in both cases and we managed to filter the sensitive data being accessed by the applications.

[3] https://github.com/Evilong/SimpleSMSApplication



(a) Full SMS List (b) Contact Specific SMS List

Figure 4.3: Simple SMS Application Screenshots

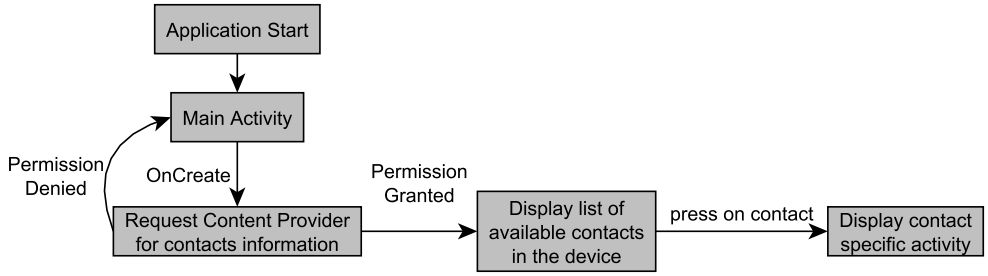


Figure 4.4: Simple Contacts Application life-cycle

4.3 Simple Wifi Mac Address Application

Recently an article exposed multiple sources for data leakage in Android [32], and one of them could be the usage of network MAC address as an hardware-based persistent identifier to better target advertisements. As such we decided to test if our implementation would be able to change our perceived network MAC address. As with the previous tests, we started by creating an application[4] that could get our address. In order to find this information we found two methods that can be used, the WifiInfo object that hold the method getMacAddress() and through listing the of NetworkInterfaces and then querying them with getHardwareAddress(). We decided on the second method as the first one was updated in Android 6.0 for security reasons to return a constant value of 02:00:00:00:00:00.

[4] https://github.com/Evilong/SimpleWifiMacAddressApplication

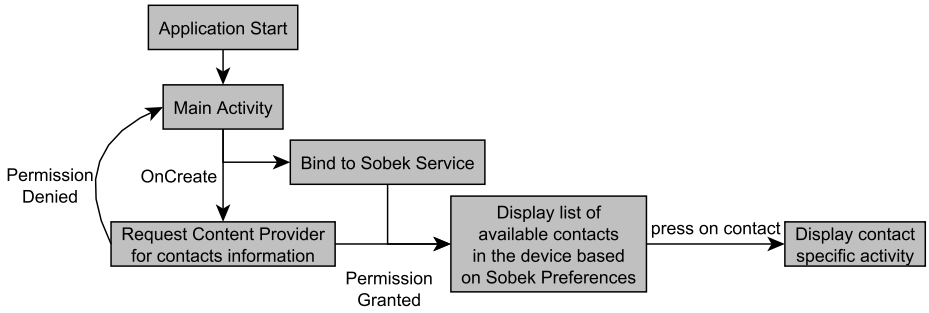


Figure 4.5: Simple Contacts Application instrumented life-cycle

The aspects used on this test targeted the very specific method of getHardwareAddress() from the NetworkInterface class. After instrumenting this application we successfully changed the behaviour of the application and could reliably change our perceived MAC address.

4.4 Facebook applications

With Facebook being one of the most popular social network [21], and being involved in the early 2018 scandal of data leakage together with Cambridge Analytica [29], their applications raise high concerns on the subject of privacy, as such we identified them as good candidates to test our implementation. The chosen applications were the Facebook, Messenger, Instagram and their Lite versions. The Lite versions are smaller and better optimized applications that offer the main features of these social network while stripping down lesser features such as animations.

We started by reviewing the permissions required by these applications and concluded that all used the same permissions we used on our sample applications. As such we decided to use the previously tested aspects.

The Table 4.1 reflects the behaviour of each application on the various tools used in our process. We concluded that both Facebook Lite and Instagram Lite applications were able to be instrumented with success, however with further analysis we uncovered most of their code base is built using native code and out of reach to our tool. With this result we also used a new aspect with success, that instruments the onCreate() function of the Application class and outputs a debug message through the usage of the Android logging API. The Facebook and Messenger applications were unable to go through our instrumentation process as they broke at the first step, the resource extraction with use of *Apktool*. In order to find our the behaviour from *dex2jar* and *enjarify* we resorted to the usage of these tools as standalone and both resulted in unusable code. Messenger Lite and Instragram even though were able to go the whole instrumentation process resulted in broken code related with the usage of native code. To pinpoint the source of this problem both of these applications were tested without code modifications and had the same result. With this we concluded that the broken code was derived from the decompilation process within *dex2jar* and *enjarify*.

Application \ Tool	Apktool	Dex2jar/Enjarify	javac + ajc
Facebook	Unable to extract resources	Unusable code	Not reached
Facebook Lite	✓	✓	✓
Messenger	Unable to extract resources	Unusable code	Not reached
Messenger Lite	✓	Broken JNI related code	✓
Instragram	✓	Broken startup due to native code	✓
Instragram Lite	✓	✓	✓

Table 4.1: Facebook owned application results

4.5 Twitter & Twitter Lite

Twitter in an widely used social network to express personal opinions, and as the previous found its way into their user's devices through two application, the Twitter and Twitter Lite. The main Twitter application, revealed a new problem while trying to apply our aspects to the code with AJC. The AJC compiler crashes while weaving and throws an org.aspectj.weaver.BCException and we were unable to track down the issue behind this problem.

On the other hand the Twitter Lite had a completely different behaviour when we applied our instrumentation. When using our tool with *dex2jar* it did not show any problems, however when installed on a device and executed it would crash right at start due to broken code associated with Firebase, an extremely popular back-end solution that helps mobile and web developers to produce high quality applications with tons of features that might require complicated server side logic. With *enjarify*, as with *dex2jar*, the tool works without revealing any concerning errors. After installation, the application works as normal, and by reviewing the logs we can conclude that instrumentation on the activities/fragments was successful. After rebooting the application it no longer works due to a crash related with broken code from the usage of retrofit2, an HTTP client for Android and Java. By inspecting the result code from enjarify, we found out that this application uses Chrome Web Client, meaning that the most of the code is not Java but Javascript.

4.6 Waze

Waze is an application that provides various utilities related with maps such as place discovery or GPS enabled routing. Outside of Google Maps which is distributed with the Operating

System (OS) Waze is one of the most complete applications in this category, as such we decided to test our implementation in it. Our objective in this test was to manage to fake our position as we did in our Simple Location Application, described in Subsection 4.1, however we were unable to do it as even though the instrumentation process in our tool worked without any compromise, the application revealed a crash a few seconds after starting up due to broken code related with the Java Native Interface (JNI).

4.7 YinzCam applications

YinzCam applications caught our attention as we identified a possible exploit within a particular feature, the ability to reproduce replays of certain moments within sports games [27].

Through usage of one of these applications he determined that these application would provide the user with this feature if the user was within a range of the location where the game is happening. With this in mind we targeted it with our location faking aspect. The instrumentation tool result was flawless with the use of *dex2jar*, as it did caught the methods and did not throw any major errors/warning that could be considered as a possible problem.

We considered this test as a success as even though at time of testing the functionalities that use the instrumented methods were not available, as they are only enabled during each sport season, because through the usage of Android *Logcat* tool we were able to find leftover debug messages that reported several tries to find out our location and were behaving accordingly to our preferences established on our user application (Section 3.2).

With the usage of *enjarify* as alternative to *dex2jar* the results were a bit different since even though the instrumentation would work without any relevant concerns, code related with Retrofit2, a popular library of HTTP in Android, would be broken.

4.8 Taxonomy

In order to understand the results, we decided to categorize the tested applications with a taxonomy. This taxonomy illustrated in the Figure 4.6 aggregates the applications in three distinct classes: **Java** where the application code is written in this language and uses mainly Android SDK as base to be built, **Native** that although the starting point for the application is through Java everything else is derived from code in C/C++ and utilizes Android Native Development Kit (NDK) as base instead of Android SDK and lastly **Java + Native** that mixes both of the previous techniques. With this categorization we are able to determine the working scope for our implementation and establish that the success rate of the instrumentation is heavily influenced by in which category the application belongs, being more successful in the category of **Java** followed by **Java + Native** and lastly not working on **Native**.

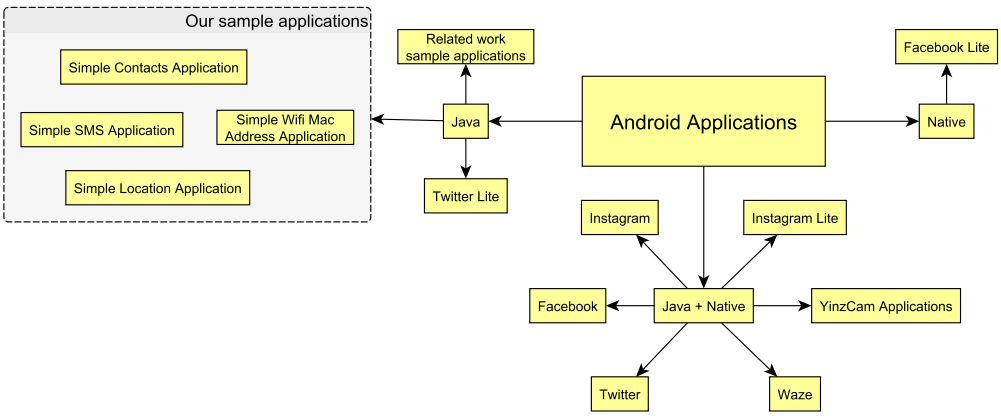


Figure 4.6: Taxonomy of Android applications

4.9 Related work evaluation

As stated in Subsection 1.2.1 we had the objective to identify the progress done to combat the leakage of sensitive data from Android devices. We reviewed various implementations that used introspection and evaluated them in order to understand the maturity of the technique. Table 4.2 illustrates the scale used to evaluate these implementations based on two factors, degree of usability and reliability of affecting applications.

level	degree of usability	reliability of affecting applications
0	not available	cannot be tested / did not work
1	build from source	simple applications
2	download from website	publicly available applications
3	available at play-store	popular applications

Table 4.2: Evaluation scale

The Figure 4.7 depicts how each one of the solutions scored against our scale.

With these results, we concluded that the current tools using introspection to protect the user are not suited for day-on-day use and usually require the users to have a high degree of knowledge of the platform.

4.10 Smartphone performance impact analysis

Our solution aims to provide better control over the applications installed in the smartphone, however we also consider that the usage of these applications should not be hindered by performance degradation.

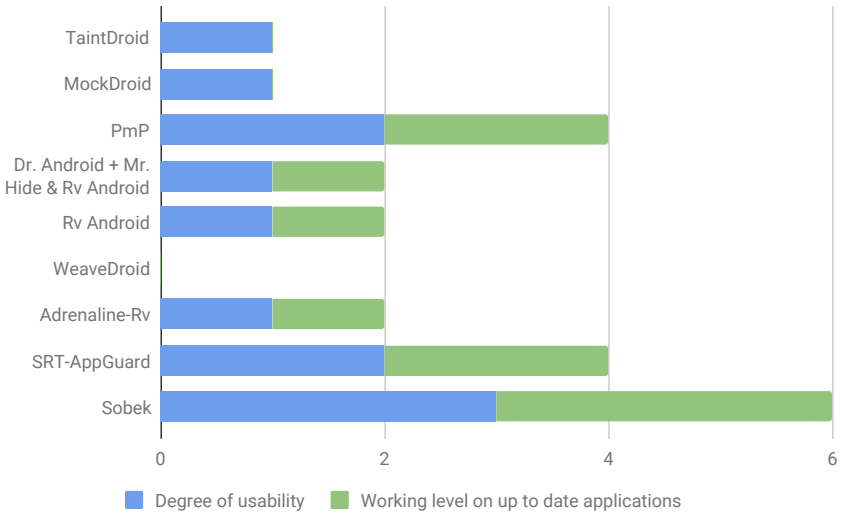


Figure 4.7: Scores based on the availability and usability for each implementation

4.10.1 Methodology and experimental setup

We started by selecting two of our sample applications as we know their expected behaviour and thus able to infer any changes on the Central Processing Unit (CPU) and Random Access Memory (RAM) usage with the respective computations while on other applications we might get changes in these values and cannot discern the reasons. The selected applications were the Simple Location Application (Section 4.1) and Simple Wifi Mac Address (Section 4.3) as these represent two distinct workflows as one needs to retrieve location from GPS and the other has to connect with the network interface. We installed two versions of each one of these applications, one normal and one instrumented on a LG Nexus 5X smartphone with Android 7.1.2 and connected to a device to a computer with Snapdragon Profiler. We ran each test ten times, and in each test we used a 1 second of delay for each measurement. The network behaviour was not recorded as none of these applications had this kind of behaviour.

4.10.2 Instrumented applications

In order to have a better understanding of the instrumented applications behaviour we decided to plot the collected data. We decided to not include the RAM usage as it was in all cases a steadily line at about 80% utilization with no variance over time.

In the Figure 4.8 and Figure 4.9 we can see the CPU utilization results for three tests in both normal and instrumented versions of the Simple Location Application. In all cases we can see three spikes with 10000 milliseconds of intervals which represent the update location function of the application. As the values of these spikes in all cases are in the same range of 10% to 15% we can conclude that our instrumentation process did not alter the application in any significant way that may impact the original performance.

In Figure 4.10 and Figure 4.11 are represented the results for the tests conducted with Simple Wifi Application and we can see three periods of similar behaviour that translate the click of the refresh button, triggering the application to try to access the network interface of the device. As with the previous test we can conclude that our instrumentation process did not have impact on the performance as the periods have similar behaviour both in normal and instrumented version.

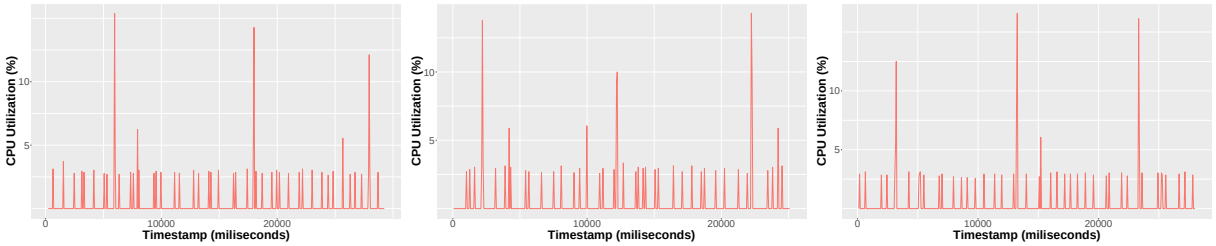


Figure 4.8: Simple Location Application performance

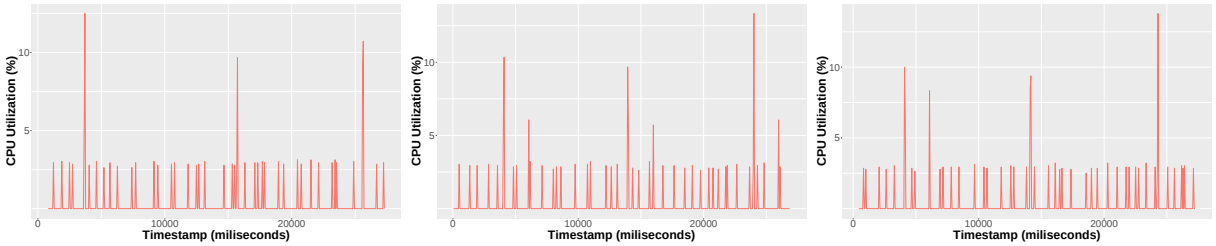


Figure 4.9: Instrumented Simple Location Application performance

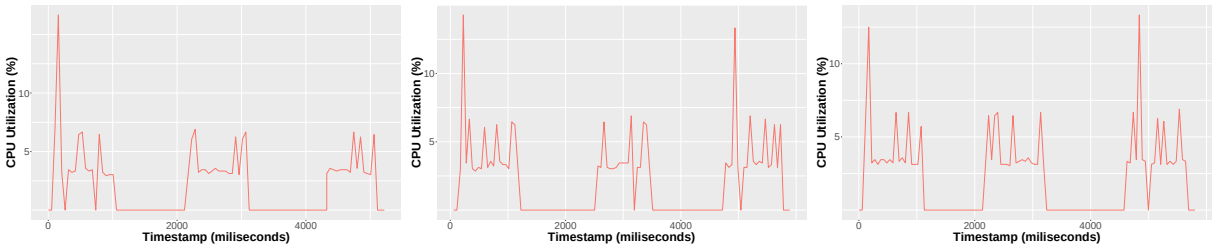


Figure 4.10: Simple Wifi Mac Address Application performance

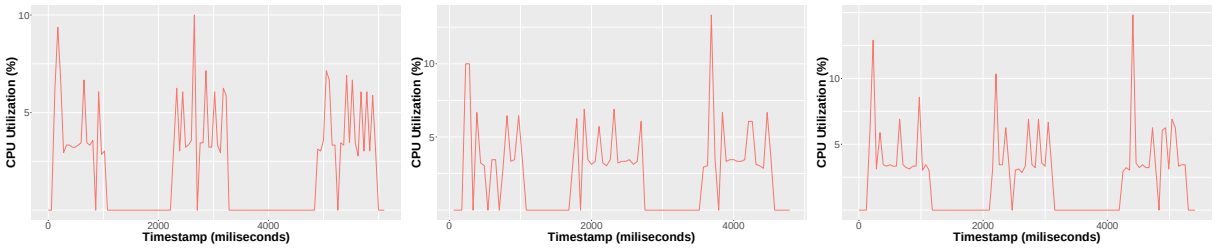


Figure 4.11: Instrumented Simple Wifi Mac Address Application performance

4.11 Back-end performance

As we aim to have a scalable implementation, we need to learn how it behaves and what is the best configuration to be deployed, thus in this section we describe the methodology used for testing both decompilers, compilers and the full implementation.

4.11.1 Methodology and experimental setup

We used a standard computer equipped with a Ryzen 3700X CPU capable of achieving 4.4 GHz while boosting on 8 cores with 16 threads and 2x16GB of RAM at 3600 MHz CL16 running Ubuntu 16.04.4 LTS (xenial) OS, as it would be simpler due to better access and cost reductions. The recorded times were obtained by using the said tools with the Linux command *time*[5] and recording the "real" value outputted ten times. The collection of this data was done back-to-back through the usage of a bash script looping for each application all the different executions. In the plots we decided to use logarithmic scale due the wide range of values obtained and the error bars represent the standard deviation to indicate the variability of the values.

4.11.2 Enjarify & Dex2jar

Enjarify is a script built in Python language and as such there are multiple interpreters. Although the author of this tool recommends the usage of *PyPy* interpreter instead of *CPython* we tested both in order to determine the best case for our setup with performance and results in mind. The Figure 4.12 shows the results of these tests and we concluded that *PyPy* performed better and that as the application size increase *CPython* gets further behind.

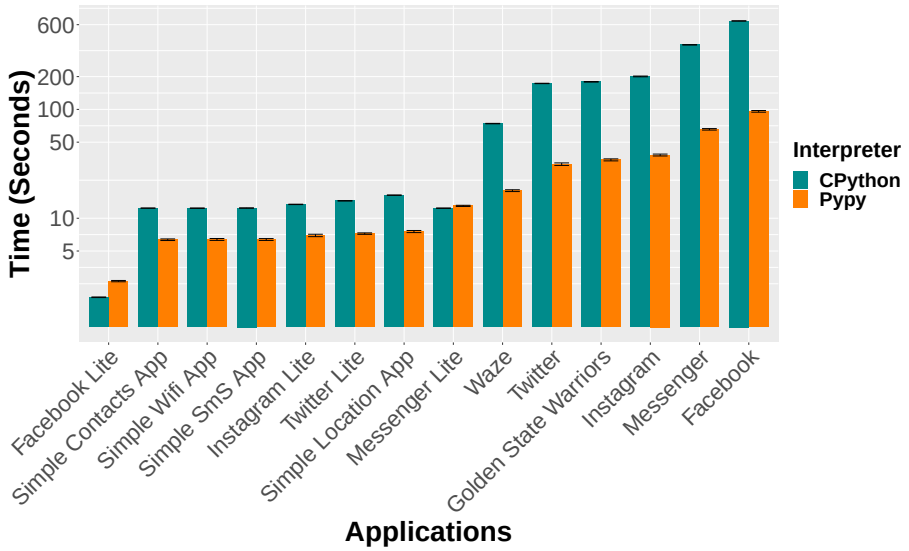


Figure 4.12: Enjarify execution time with *PyPy* and *CPython* interpreters

[5]time - run programs and summarize system resource usage

After these tests we decided to see how *Enjarfiy* performs in various types of applications and as it outputs the number of classes processed we used it in order to plot with the execution time in the Figure 4.13. We find that the runtime varies linearly with the number of classes processed. An additional information we recorded in these tests was that some produced results although with classes containing errors. These cases were Messenger, Instagram, Twitter and Golden State Warriors applications with three, four, three and ten errors respectively.

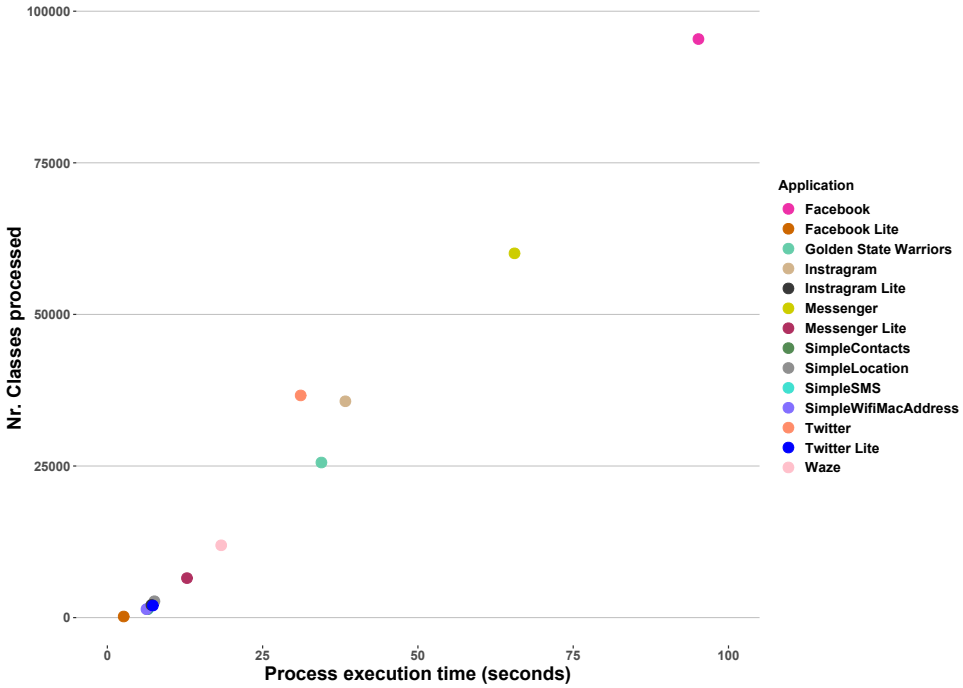


Figure 4.13: Enjarify execution time per classes processed

dex2jar allows different allocations of memory, and by default uses 1gb of RAM. While testing this tool, we found out that it would crash due to the lack of memory. As such we decided to test different sizes of memory allocation in order to solve these crashes and establish a reliable value for most of the applications. The Figure 4.14 illustrates the results of these tests. All times are recorded for full execution including when the execution would crash (Facebook, Messenger, Golden State Warriors). The large standard deviation variability in Golden State Warrior and Messenger values when running at 1 GB is derived from unpredictable behavior, crashing sometimes after long runs or with successful runs on small runs. These results also show that the allocated memory does not impact the execution time of the process, however it allows further progress in some applications. Unfortunately this progress does not translate to a successful result as it would crash but not due to running out of memory (Facebook, Messenger).

While implementing the solution, we decided to plot the performance of *enjarify* against *dex2jar* while decompiling an application from their original .apk file in order to have a better understanding on the behaviour of these tools and have an educated guess on time required for their executions.

The Figure 4.15 illustrates the results of these tests, and we are able to conclude the following:

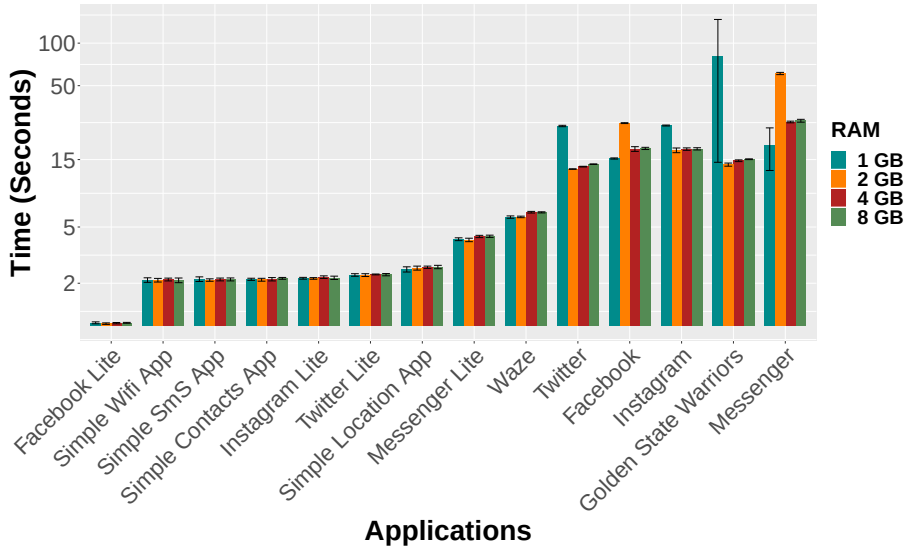


Figure 4.14: Dex2jar execution with different sizes of memory allocation performance

- The bigger the application code base is the more time is needed to decompile
- *dex2jar* running with 8gb of RAM is faster than *enjarify*
- *enjarify* while being slower was able to complete its execution in contrast to *dex2jar* that failed both Facebook and Messenger application.

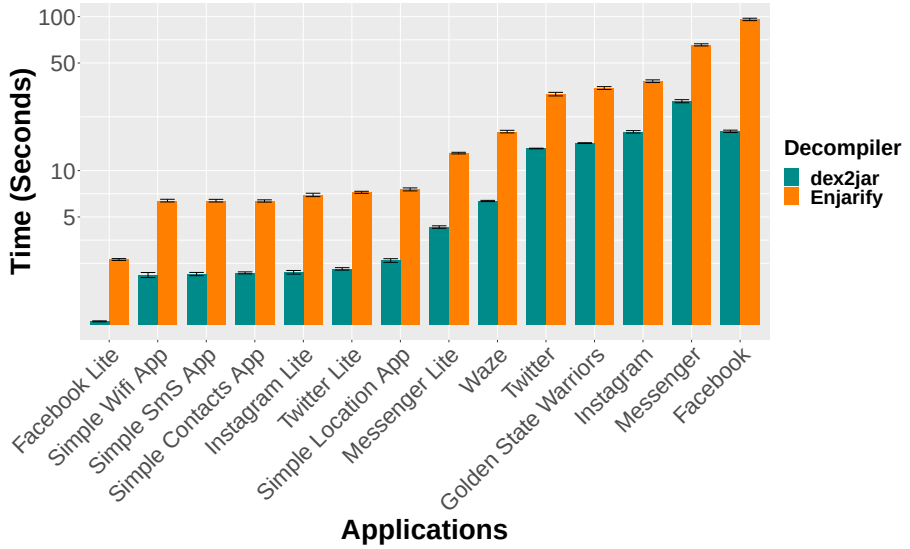


Figure 4.15: Enjarify vs Dex2jar performance

4.11.3 Sobek Instrumentation Tool

The Figure 4.16 illustrates the performance of the Sobek Instrumentation Tool (Section 3.1), while instrumenting our selected applications described previously. In this figure, Facebook

application is missing as it can not be instrumented by our tool, as *Apktool* is not able do unpackage its files. Through analisis of these results we can conclude that the difference of execution times between compilers, *dx* and *r8*, is significant and that *dx* is faster in most of the cases. However as mentioned earlier in Section 3.1, *dx* has lower chances of producing usable applications.

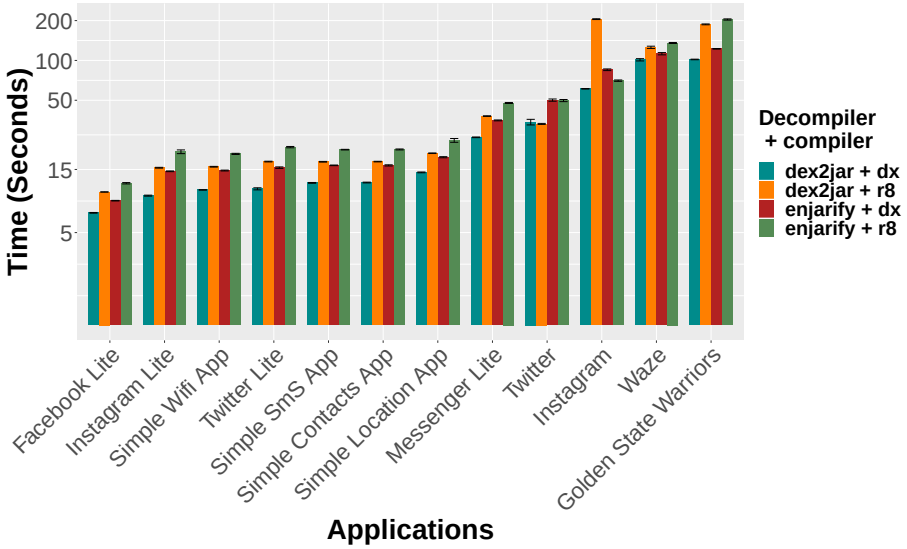


Figure 4.16: Sobek Instrumentation Tool performance

Afterwards we decided to profile the execution of the tool with the worse case (*Enjarify* + *r8* while instrumenting the Golden State Warriors application) in order to have a better understanding on how we should setup Kubernetes. This profiling was done by monitoring both CPU and RAM usage. In the Figure 4.17 we represent the RAM usage with dotted lines and the other the CPU utilization. We can clearly see the various phases of our instrumentation, each representing a new peak in both measurements, and we conclude that the highest performing task is to recompile code that reaches 100% CPU utilization.

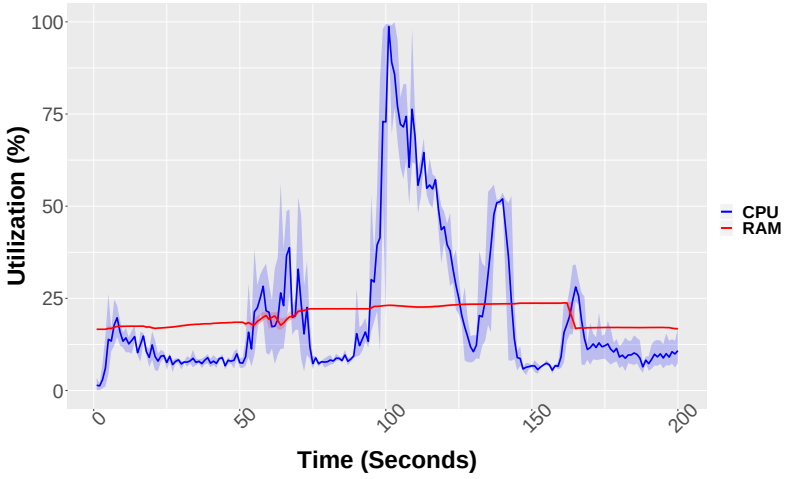


Figure 4.17: Sobek Instrumentation Tool execution profile

Chapter 5

Conclusion

In this thesis, we documented the process of creating a functional and modular instrumentation system for Android applications able to make severe improvements on the data confidentiality and privacy on these devices as determined in Subsection 1.2.1.

We started by reviewing the current literature in Section 2.5 and found out that the most of the of these experiments are unusable by simple users or are not able to be used in up-to-date application as described in Section 4.9.

In the end, we created and successfully applied an implementation based on introspection techniques and we can reliably lie to simple applications that targeted personal data.

In summary we were able to:

- Review similar implementation and provide an overview of their current status.
- Create a functional prototype of the proposed security system, including a back-end and a manager application.
- Prove that Aspect-Oriented Programming (AOP) can be used on Android applications.
- Use AOP successfully to modify code that accesses personal data.
- Instrument applications without impacting their performance.
- Modify application behaviour without needed to root the device.
- Determine current implementation limitations (Section 5.1).
- Propose insight on how the overcome the limitations found (Section 5.2).

5.1 Current implementation limitations

While researching on the subject we also had a glimpse of several problems that may limit the previous implementations and our implementation. *dex2jar* and *enjarify* were tools created with the intent to help researchers to reverse-engineer code, however it this code is not supposed to be subject to extensive modifications and recompiled back.

With *dex2jar*, the maturity of the tool shows that even though it is not actively developed anymore it still can produce usable results sometimes. Many of the issues encountered derive from the lack of support for Java 7/8/9 features. Another cause for bad code is that obfuscation tools got much better since the time where it had active development.

Enjarify, tries to be better than *dex2jar* as it was built while knowing the problems it faced. Although even though the decompiled code is better for human understanding it fails many times to produce compilable/working code. One specific reason for this the lack of support for Java annotations which are popular in the Android development scene [36] that revealed itself as a problem during our tests to Yinzcam application described in Section 4.7.

Apktool revealed itself as a great tool for the whole process, even though it performed as expected we need to remark that it failed in one specific case, the Facebook application.

We consider that one of the limitations of our implementation was the use of *AspectJ*. This implementation of AOP was built to instrument Java code and as seen in Section 4.8 applications may be built entirely or partially with C/C++ code.

Although not a limitation per se as we managed to work around it, the default dex compiler is not suited for this kind of work as it is unable to compile the most of the instrumented code.

5.2 Future work & open research challenges

The limitations described previously in Section 5.1 make implementations based on these techniques not suited for day-to-day used. As such we propose a few ideas on what can be done in order to solve some of these problems:

- Research and experiment with tools capable of instrumenting the C/C++ native code in order to reach all languages used in the Android platform.
- Explore C/C++ decompilers and add module based on these findings to have better coverage of the supported languages on Android.
- Create a Dex decompiler with a purpose to produce instrumentable code. This decompiler should aim to generate code that may not be human-readable but should retain all the original features.

While not problems, we also have ideas on some improvements to the implementation that would make it more resilient such as:

- Verification of produced results through automated tests on the new instrumented application.
- Analysis of application before instrumenting in order to provide a better selection of used processes.
- Create a front-end in order to interact with the REST API (Subsection 3.3.6) providing the user with a user-friendly interface to manage their settings remotely.
- Utilization of smart anonymization techniques in the fake information to reduce the input needed from the user.

Appendix A

Development notes

A.1 Programming languages used

- Bash - Sobek Instrumentation Tool shell-script
- Go - gRPC-gateway implementation
- Groovy - Gradle build files for Android applications and backend
- Java - Android applications, backend implementation and instrumentation aspects
- Structured Query Language (SQL) - Used to create and interact with Sobek Database

A.2 Data definition languages used

- Android Interface Definition Language (AIDL) - Definition of Sobek Inter-Process Communication (IPC) service
- Comma-separated values (CSV) - Storing information and data transfer in Android applications
- JavaScript Object Notation (JSON) - Storing information such as credentials and definitions for Sobek Backend Services. Also present in OpenAPI definition.
- Protobuf - Defining gRPC protocol for communication between mobile application and backend
- Extensible Markup Language (XML) - Android resource files
- YAML Ain't Markup Language (YAML) - Docker and Kubernetes instructions files

A.3 Software used

- **Apktool**
  - Version: 2.4.0
  - Website: https://github.com/iBotPeaches/Apktool
- **Enjarify**
  - Version: 1.0.3
  - Website: https://github.com/Storyyeller/enjarify
- **dex2jar**
  - Version: 2.1
  - Website: https://github.com/pxb1988/dex2jar
- **Android Software Development Kit (SDK)**
  - Version: 28
  - Website: https://www.jetbrains.com/idea/
- **Java Standard Edition (SE) Java Development Kit (JDK)**
  - Version: 1.8.0
  - Website: https://www.oracle.com/technetwork/java/javase
- **MySQL**
  - Version: 2nd Gen 5.7
  - Website: https://dev.mysql.com/downloads/mysql/5.7.html
- **Docker**
  - Version: 18.09.2
  - Website: https://www.docker.com/
- **Google Kubernetes Engine**
  - Version: 1.12.7-gke.25
  - Website: https://cloud.google.com/kubernetes-engine/
- **r8**
  - Version: 1.6.0
  - Website: https://r8.googlesource.com/r8

- **protoc**
  - Version: 2.6.1
  - Website: https://github.com/protocolbuffers/protobuf
- **grpc-gateway**
  - Version: 1.8.0
  - Website: https://github.com/grpc-ecosystem/grpc-gateway
- **AspectJ**
  - Version: 1.9.3
  - Website: https://www.eclipse.org/aspectj/
- **PyPy**
  - Version: 7.0.0
  - Website: http://pypy.org/
- **CPython**
  - Version: 3.5.2
  - Website: https://github.com/python/cpython

A.4 Utilities used

- **IntelliJ IDEA**
  - Description: An Java Integrated Development Environment (IDE).
  - Usage: Develop the Sobek Backend services.
  - Version: Ultimate Edition - 2018.3.5 -> 2019.1
  - Website: https://www.jetbrains.com/idea/
- **Android Studio**
  - Description: An Java IDE with special focus on Android development tools and features.
  - Usage: Build Sobek Manger and test apps, also to conduct tests on Android devices.
  - Version: 3.2 -> 3.4
  - Website: https://developer.android.com/studio

- **Visual Studio Code**
  - Description: A source-code editor able to work with the most of the programming language through the high customization available with plugins.
  - Usage: The main source-code editor for pieces of code developed outside of an IDE.
  - Version: 1.28.1 -> 1.38.1
  - Website: https://code.visualstudio.com/
- **DBeaver**
  - Description: An universal database tool that simplifies the most of the tasks related through an user friendly Graphical User Interface (GUI).
  - Usage: Create and modify the Sobek database.
  - Version: Community Edition - 6.0.1
  - Website: https://dbeaver.io/
- **Snapdragon Profiler**
  - Description: Profiling software for Android devices powered by Snapdragon processors.
  - Usage: Monitor and record Central Processing Unit (CPU) and Random Access Memory (RAM) for smartphone performace impact analysis tests
  - Version: 2019.2.0.7022019
  - Website: https://developer.qualcomm.com/software/snapdragon-profiler
- **Google Sheets**
  - Description: A web based spreadsheet program.
  - Usage: Analysis of the performance tests data and graph creation.
  - Website: https://www.google.com/sheets/about/
- **R Studio**
  - Description: An IDE for for R programming language.
  - Usage: Analyse performance tests data and create graphs to convey their results.
  - Version: 3.5.1
  - Website: https://www.rstudio.com/
- **Adobe Photoshop**
  - Description: An feature rich graphic editor.
  - Usage: Creation of Sobek Manager application icon and prototype its User Interface (UI).
  - Version: CC 2018
  - Website: https://www.adobe.com/products/photoshop.html

Appendix B

Code snippets

```
/*
Messages definition for DeviceManagementService
*/
message AndroidId {
    string id = 1;
}
message ServerInformationOnDevice {
    google.protobuf.Timestamp lastPreferenceChange = 1;
    google.protobuf.Timestamp lastPreferenceSync = 2;
}
message DeviceAccount{
    string id = 1;
    string accountid = 2;
}
message DevicesList {
    map<string,ServerInformationOnDevice> devicesList = 1;
}
message Preference {
    string key = 1;
    string value = 2;
}
message Preferences {
    repeated Preference preferences = 1;
}
message UpdateGeneralPreferences {
    string id = 1;
    repeated Preference preferences = 2;
}
message ApplicationSpecificPreferences {
    string id = 1;
    string packageName = 2;
}
message UpdateApplicationSpecificPreferences {
    string id = 1;
    string packageName = 2;
```

```
36      repeated Preference preferences = 3;
37  }
38  message LogsList {
39      map<string, google.protobuf.Timestamp> logsList = 1;
40  }
41  message LogEntry {
42      google.protobuf.Timestamp time = 1;
43      string message = 2;
44  }
45  message ApplicationLogForDevice {
46      string id = 1;
47      string packageName = 2;
48  }
49  message ApplicationLog {
50      repeated LogEntry entries = 1;
51  }
52  message UpdateLog {
53      string id = 1;
54      string packageName = 2;
55      repeated LogEntry entries = 3;
56  }
57  message Location {
58      string name = 1;
59      double latitude = 2;
60      double longitude = 3;
61  }
62  message Locations {
63      map<int32,Location> locations = 1;
64  }
65  message LocationUpdate {
66      string id = 1;
67      map<int32,Location> locations = 2;
68  }
69
70  service DeviceManagementService {
71      rpc getDevices (google.protobuf.Empty) returns (DevicesList) {
72          option (google.api.http) = {
73              get: "/device"
74          };
75      }
76      ↩
        rpc getInformationOnDevice (AndroidId) returns (ServerInformationOnDevice) {
77          option (google.api.http) = {
78              get: "/device/{id}"
79          };
80      }
81      rpc createDeviceOnServer (DeviceAccount) returns (OperationResult) {
82          option (google.api.http) = {
83              post: "/device/{id}"
84              body: "*"
85          };
```

```
86      }
87      rpc readDevicePreferences (AndroidId) returns (Preferences) {
88          option (google.api.http) = {
89              get: "/device/{id}/preferences"
90          };
91      }
92      ↩
        rpc writeDevicePreferences(UpdateGeneralPreferences)returns(OperationResult){
93          option (google.api.http) = {
94              put: "/device/{id}/preferences"
95              body: "*"
96          };
97      }
98      rpc readDevicePreferencesForApplication (ApplicationSpecificPreferences)
99      returns (Preferences) {
100         option (google.api.http) = {
101             get: "/device/{id}/preferences/{packageName}"
102         };
103     }
104     ↩
        rpc writeDevicePreferencesForApplication(UpdateApplicationSpecificPreferences)
105     returns (OperationResult) {
106         option (google.api.http) = {
107             put: "/device/{id}/preferences/{packageName}"
108             body: "*"
109         };
110     }
111     rpc listDeviceLogs (AndroidId) returns (LogsList) {
112         option (google.api.http) = {
113             get: "/device/{id}/logs"
114         };
115     }
116     rpc readDeviceLog (ApplicationLogForDevice) returns (ApplicationLog) {
117         option (google.api.http) = {
118             get: "/device/{id}/logs/{packageName}"
119         };
120     }
121     rpc writeDeviceLog (UpdateLog) returns (OperationResult) {
122         option (google.api.http) = {
123             put: "/device/{id}/logs/{packageName}"
124             body: "*"
125         };
126     }
127     rpc readDeviceLocations (AndroidId) returns (Locations) {
128         option (google.api.http) = {
129             get: "/device/{id}/locations"
130         };
131     }
132     rpc writeDeviceLocations (LocationUpdate) returns (OperationResult) {
133         option (google.api.http) = {
134             put: "/device/{id}/locations"
```

```
135             body: "*"
136         };
137     }
138 }
```

Listing B.1: Protobuf Device Managment Service Definition

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import android.util.Log;

import android.content.Intent;
import android.content.Context;
import android.content.ServiceConnection;
import android.content.ComponentName;
import android.os.IBinder;
import android.net.Uri;
import android.app.Activity;
import android.app.Fragment;
import android.app.Application;
import android.content.Context;
import android.net.wifi.WifiInfo;

import pt.up.fc.dcc.sobekmanager.ISobekAidlInterface;
import java.util.Arrays;
import java.lang.reflect.*;
import android.database.Cursor;
import android.provider.ContactsContract;
import java.util.concurrent.Semaphore;

@Aspect
public class WifiInfoAspect {

    boolean mIsBound;
    protected ISobekAidlInterface sobekServer;
    public Activity activity;

    private ServiceConnection serviceConnection = new
        ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName componentName,
            IBinder iBinder) {
            Log.d("SobekInstrumentation", "Service Connected");
            sobekServer = ISobekAidlInterface.Stub.asInterface(iBinder);
        }
```

```
        @Override
        public void onServiceDisconnected(ComponentName componentName) {
            Log.d("SobekInstrumentation", "Service Disconnected");
            sobekServer = null;
        }
    };

    void doBindService() {
        Log.d("SobekInstrumentation", "Binding service");
        Intent i = new Intent();
        i.setComponent(new ComponentName("pt.up.fc.dcc.sobekmanager",
            "pt.up.fc.dcc.sobekmanager.services.PermissionsService"));
        activity.bindService(i,
           serviceConnection, Context.BIND_AUTO_CREATE);
        mIsBound = true;
    }

    void doUnbindService() {
        Log.d("SobekInstrumentation", "Unbinding service");
        if (mIsBound) {
            activity.unbindService(serviceConnection);
            mIsBound = false;
        }
    }

    @Pointcut("(call(void *.onCreate(..))
        || call(void *.onResume(..))) && this(android.app.Activity+)")
    public void onActivityCreateEntryPoint() {
    }

    @After("onActivityCreateEntryPoint()")
    public void setActivityPointcut(JoinPoint jp) {
        Log.d("SobekInstrumentation", "New activity");
        activity = (Activity) jp.getThis();
        doBindService();
        return;
    }

    @Pointcut("(call(void *.onCreate(..)) ||
        call(void *.onResume(..))) && this(android.app.Fragment)")
    public void onFragmentCreateEntryPoint() {
    }

    @After("onFragmentCreateEntryPoint()")
    public void setActivityfromFragmentPointcut(JoinPoint jp) {
        Log.d("SobekInstrumentation", "New fragment");
```

```
        Fragment frag = (Fragment) jp.getThis();
        activity = frag.getActivity();
        doBindService();
        return;
    }

    @Pointcut("call(void *.onDestroy(..))")
    public void onActivityDestroyEntryPoint() {
    }

    @Around("onActivityDestroyEntryPoint()")
    public Object destroyActivity(ProceedingJoinPoint jp) throws
        Throwable {
        Log.d("SobekInstrumentation", "Current activity destroyed");
        doUnbindService();
        return jp.proceed();
    }

    @Pointcut("call(* getHardwareAddress())")
    public void getMacAddressFromNetworkInterfaceEntryPoint() {
    }

    @Around("getMacAddressFromNetworkInterfaceEntryPoint()")
    public Object changeGetMacAddressFromNetworkInterfaceAround
        (ProceedingJoinPoint pjp) throws Throwable {
        if (activity != null && mIsBound && sobekServer != null) {
            String accessType =
                sobekServer.accessType(activity.getApplicationContext()
                .getPackageName(), "wifiMacAddress");
            if (accessType == null || accessType.isEmpty()
                || accessType.equals("Allow"))
                return pjp.proceed();
            if (accessType.equals("Fake")) {
                Log.d("SobekInstrumentation", "Changing
                    wifiMacAddress");
                String fakeMac = sobekServer
                        .fakeMacAddress(activity.getApplicationContext()
                        .getPackageName());
                if (fakeMac == null)
                    return pjp.proceed();
                return fakeMac.getBytes();
            }
            return pjp.proceed();
        } else {
            Log.d("SobekInstrumentation", activity +
                " " + mIsBound + " " + sobekServer);
            Log.d("SobekInstrumentation", "Tried to change getMacAddress
```

```
                but activity was null");
            return pjp.proceed();
        }
    }
}
```

Listing B.2: Wifi Information Aspect

Bibliography

[1] The Kubernetes Authors. Horizontal Pod Autoscaler - Kubernetes, 2019. Last visited 2019-07-08.

[2] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–fine-grained policy enforcement for untrusted android applications. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 213–231. Springer, 2014.

[3] Hakan Bagci and Ahmet Kara. A lightweight and high performance remote procedure call framework for cross platform communication. In *ICSOFT-EA*, pages 117–124, 2016.

[4] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th workshop on mobile computing systems and applications*, pages 49–54. ACM, 2011.

[5] Frank Buschmann. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, 2011.

[6] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I Hong, and Yuvraj Agarwal. Does this app really need my location?: Context-aware privacy management for smartphones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3): 42, 2017.

[7] Stelvio Cimato, Alfredo Santis, and Umberto Petrillo. Overcoming the obfuscation of java programs by identifier renaming. *Journal of Systems and Software*, 78:60–72, 10 2005. doi:10.1016/j.jss.2004.11.019.

[8] Philip Daian, Ylies Falcone, Patrick Meredith, Traian Florin Şerbănuţă, Akihito Iwai, Grigore Rosu, et al. Rv-android: Efficient parametric android runtime verification, a brief tutorial. In *Runtime Verification*, pages 342–357. Springer, 2015.

[9] Corbin Davenport. Google bans crypto miners, 'repetitive content,' and more from the Play Store, 2018. Last visited 2016-07-08.

[10] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[11] 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.

[12] Yliès Falcone and Sebastian Currea. Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 350–353. ACM, 2012.

[13] Google. Android d8, . Last visited 2019-07-10.

[14] Google. Missing Java Annotations, . Last visited 2016-07-08.

[15] Google. Android r8 repository, . Last visited 2019-07-10.

[16] Google. Android 6.0 Marshmallow, 2015. Last visited 2018-11-24.

[17] Google. Android Platform Architecture, 2018. Last visited 2018-12-02.

[18] Google. Kubernetes, 2018. Last visited 2018-12-02.

[19] Google. App permissions best practices, 2018. Last visited 2018-12-16.

[20] grpc ecosystem. gRPC-gateway Github page, 2019. Last visited 2019-09-10.

[21] David John Hughes, Moss Rowe, Mark Batey, and Andrew Lee. A tale of two sites: Twitter vs. facebook and the personality predictors of social media usage. *Computers in Human Behavior*, 28(2):561–569, 2012.

[22] Daniel Ionescu. Google Play Grows Up: New Developer Policies Will Clean Up Google's App Store, 2012. Last visited 2016-07-08.

[23] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[24] Anthony D JoSEP, RAnDy KAtz, AnDy KonWinSKi, LEE Gunho, DAViD PAttERSon, and ARiEL RABKin. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[25] Hsiangchu Lai, Jack Shih-Chieh Hsu, and Min-Xun Wu. The impact s of requested permission on mobile app adoption: The insights based on an experiment in taiwan. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[26] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250. ACM, 2012.

[27] Nathan D Mickulicz, Priya Narasimhan, and Rajeev Gandhi. Yinzcam: Experiences with in-venue mobile video and replays. In *Presented as part of the 27th Large Installation System Administration Conference ({LISA} 13)*, pages 133–144, 2013.

[28] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[29] Susanna Paasonen. Affect, data, manipulation and price in social media. *Distinktion: Journal of Social Theory*, 19(2):214–229, 2018.

[30] European Parliament. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[31] Bogdan Petrovan. Google is now manually reviewing apps that are submitted to the Play Store!, 2015. Last visited 2016-07-08.

[32] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions systems. 2019.

[33] Patrick Schulz. Code protection in android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, 110, 2012.

[34] Sarah Spiekermann, Alessandro Acquisti, Rainer Böhme, and Kai-Lung Hui. The challenges of personal data markets and privacy. *Electronic markets*, 25(2):161–167, 2015.

[35] Statista. Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018, 2018. Last visited 2018-11-24.

[36] Storyyeller. Missing Java Annotations, 2012. Last visited 2016-07-08.

[37] Haiyang Sun, Andrea Rosa, Omar Javed, and Walter Binder. Adrenalin-rv: Android runtime verification using load-time weaving. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 532–539. IEEE, 2017.

[38] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 15–24. ACM, 2011.

[39] 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.

[40] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on android). In *International conference on Trust and trustworthy computing*, pages 93–107. Springer, 2011.