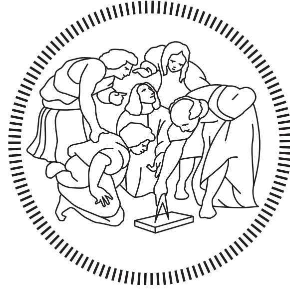
Politecnico di Milano
Scuola di Ingegneria dell'Informazione



Polo territoriale di Como
Master of Science in Computer Engineering
Communication and Society Engineering

Detecting Android malware campaigns via application similarity analysis

Supervisor: Prof. Stefano ZANERO
Co-Supervisor: Prof. Federico MAGGI

Master Graduation Thesis of:
Leonardo PRETI
Matricola N. 822875

Academic Year 2014-2015

Abstract

Due to the increasing detection of Android malware and the ever growing worldwide market share of Google's mobile operating system, which reached 84.7% in Q3 2015, the tools used to identify malicious applications must be kept updated and they need to offer a fast way to have the most reliable overview on the malware scene.

As part of the effort to mitigate this threat, we present an approach and tool that allow the security analysts to perform accurate searches on the analysed applications and find correlations between them, grouping malicious applications into clusters of similarity.

Our work is based on the notion that the most valuable asset that a malware can steal is the user's data and that, in order to retrieve them, the application must connect to an endpoint (e.g., a server, a phone) controlled by the criminal. Performing a static analysis of the source code and then comparing the information gathered with third-party dynamic tools, we obtain the list of endpoints contained in the application; if some of them are categorized as malicious, we can conclude that the application is a threat.

In order to achieve this goal, we collect general information about the application (e.g. the package name, the Android version required, the usage of Google Cloud Messaging); the components of the application (i.e., the activities, services, broadcast receivers and content providers which the application is composed of); textual information contained in the application (e.g. the strings, the URLs, the phone numbers). We enriched the collected static data by classifying the endpoints according to their maliciousness and by localising the geographical area targeted by the malware, though a deeper investigation on the strings. The comparison between applications, in order to find correlations, is performed both on the information gathered and on the similarity of the packages: if two applications are connected to the same malicious endpoint, they are under control of the same treat agent; similarly, if two applications follow the suggested guidelines and have similar package names, they have likely been developed by the same company or generated by the same crimeware kit.

The final result of our work is a web application based on Elasticsearch, a service which offers both the functionalities of an online database and of a full-text indexed search engine. Its flexibility allows us to perform queries on the samples stored and discover the clusters of similarity to which a malware belongs to in almost real-time.

Sommario

A causa dell'aumentare del numero di malware per Android rilevati e la costante crescita della fetta di mercato mondiale controllata dal sistema operativo di Google, che ha raggiunto l'84.7% nel Q3 2015, gli strumenti di identificazione di applicazioni pericolose devono essere mantenuti aggiornati e devono offrire un'affidabile visione d'insieme sul panorama criminale.

Come parte del lavoro dedicato a mitigare questa minaccia, presentiamo un'approccio e soluzione che permette agli analisti di sicurezza di effettuare accurate ricerche tra le applicazioni già analizzate e trovare correlazioni tra di esse, creando gruppi di similarità.

Il nostro lavoro è basato sulla nozione che il bene di maggior valore che un malware può rubare siano i dati degli utenti e che, per poterli ottenere, l'applicazione deve connettersi a un endpoint (e.g., un server, un telefono) sotto il controllo del criminale. Eseguendo un'analisi statica del codice sorgente dell'applicazione e confrontando le informazioni ottenute con strumenti dinamici di parte terze, otteniamo la lista degli endpoint contenuti dalla applicazione; se alcuni di essi sono categorizzati come dannosi, possiamo concludere che l'appicazione sia un pericolo.

Per potere raggiungere questo obiettivo, raccogliamo informazioni generali riguardanti l'applicazione (e.g., il package name, la versione di Android richiesta, l'utilizzo di Google Cloud Messaging); i componenti dell'applicazione (i.e., le activity, i service, i broadcast receivers e i content provider che costituiscono l'applicazione); informazioni testuali contenute dall'applicazione (e.g., le stringhe, gli URL, i numeri telefonici). Abbiamo arricchito i dati statici raccolti classificando gli endpoint in base alla loro pericolosità e localizzando il mercato di riferimento del malware, attraverso un'investigazione approfondita delle stringhe. La comparazione tra applicazioni, al fine di trovare correlazioni, è effettuata sia sulle informazioni raccolte, sia sulla similarità tra i package name: se due applicazioni sono connesse al medesimo endpoint malevolo, sono sotto controllo dello stesso criminale; se due applicazioni rispettano le linee guida suggerite e hanno i package name simili, sono state probabilmente sviluppate dalla medesima compagnia o sono state generate attraverso lo stesso crimeware kit.

Il risultato finale del nostro lavoro è una web application basato su Elasticsearch, un servizio che offre sia le funzionalità di un database online che quelle di un motore di ricerca indicizzato. La sua flessibilità ci permette di eseguire query parametrizzate e scovare i cluster di similarità a cui un malware fa parte approssimativamente in tempo reale.

Contents

1 Introduction **1**

2 Background and motivations **4**
2.1 Malware growth in Android 4
2.1.1 Android market share expansion 4
2.1.2 Google's effort to prevent malware 5
2.1.3 Banking trojans 6
2.2 DroydSeuss design and achievements 6
2.2.1 Data extraction 7
2.2.2 Data enrichment 8
2.2.3 Data storage 9
2.2.4 DroydSeuss today 9

3 Architecture **12**
3.1 Components 12
3.1.1 Legacy components 12
3.1.2 New components 13
3.2 Communication flow 14
3.2.1 Structure of the analysis 14
3.2.2 Perform a search 15
3.2.3 Find similar applications 15

4 Implementation **19**
4.1 Static analysis improvement 19
4.1.1 Extension of the extracted data 19
4.1.2 Data enhancement 20
4.2 Data persistence and retrieval 24
4.2.1 Libraries integration 24
4.2.2 Data structure 25
4.2.3 Elasticsearch usage 25
4.3 Clusters of similarity 28

4.3.1 Common endpoints 29
4.3.2 Packages similarity 30
4.4 Web application 32
4.4.1 Design choices 32
4.4.2 User interface and user experience 33

5 Experimental evaluation 36
5.1 Performance 36
5.1.1 Retrieval from the database 37
5.1.2 Find correlated applications 39
5.1.3 Overall performance conclusions 40
5.2 Use cases 42
5.2.1 Detection of a malicious campaign by package similarity 43
5.2.2 Detection of a malicious campaign by common malicious endpoints 44
5.2.3 Grouping applications of the same benign developer . . 47
5.3 Limitations 49
5.3.1 Sandbox limitations 49
5.3.2 Human supervision 49
5.3.3 Real database performance 50

6 Conclusions 51
6.1 Future work 51
6.1.1 Usage of `strings` and `classes` 51
6.1.2 Automatic detection rules 52
6.2 Conclusions 53

Appendices 57

A Elasticsearch mapping 58

B Regexen to filter endpoints 62

List of Figures

2.1 Data processing pipeline of DroydSeuss 8

3.1 Original DroydSeuss analysis sequence 16
3.2 Addition to DroydSeuss analysis sequence 17
3.3 Data flow in a search 17
3.4 Data flow in a research of clusters of similarity 18

4.1 Advanced search feature entry point 34
4.2 Search bar after a query with results found 34
4.3 Search bar after a query with no results 34
4.4 Analysis report 35
4.5 The result of a failed similarity search 35
4.6 Similarity report showing correlated applications 35

5.1 Time taken to retrieve the entire database 38
5.2 Time taken to find a cluster of similarity 41
5.3 Time required per application stored when fetching the database 43

List of Tables

2.1 DroydSeuss original database schema 10
2.2 DroydSeuss report example 11

3.1 The source of the data we employed 13

4.1 Unicode Character Ranges 23
4.2 Data translation from MongoDB to Elasticsearch 26

5.1 Average time to search over the entire database 39
5.2 Standard deviation on a search over the entire database 39
5.3 Average time to find correlated applications 42
5.4 Standard deviation to find correlated applications 42
5.5 Extract of the data in common between the 15 samples 45
5.6 Samples from malicious campaign 46
5.7 General information about the two samples 46
5.8 Extract of the content shared by the two samples 46
5.9 Samples developed by *Facebook Mobile* we considered 47
5.10 The most significant shared information extracted from *Facebook Mobile* apps 48

List of Algorithms

1 Pseudo code for IP standardization 22
2 Pseudo code of similarity finding 29
3 Pseudo code for calculation of Levenshtein distance 31

List of Listings

4.1 Elasticsearch data structure 27
4.2 Elasticsearch GET API 27
4.3 `elasticsearch-py` usage 28
4.4 `elasticsearch-dsl` usage 28
A.1 Elasticsearch mapping 58
B.1 Regex definition of a domain 62
B.2 Regex definition of an IP 63

Chapter 1

Introduction

Android is the most popular mobile operating system, surpassing in Q2 2013 75% of market share, reaching 84.7% in Q3 2015 [22] and is expected to sell 1.62 billion devices in 2020, with a compound annual growth rate of 6.9% [9]. This widespread acceptance of Google's operating system is mirrored in owning the largest app store, having 1.6 million applications available for download in the Google Play Store [21] and expecting an increase of 19.53% in worldwide apps downloads in 2017 [20]. In addition to these numbers, Android's open design allows the users to download and install applications from outside the official channel: there is an estimate of over 1 million applications spread among third-party app stores, of which 20,000 are newly released every month. Cyber criminals naturally recognized the possibility of exploiting Google's loose control over this app-distribution model by developing mobile malware, leading to a growth of 388% of malicious applications released between 2011 and 2013 [17]. Google's first answer to these threats was the introduction of Bouncer in February 2012, an automated tool to detect anomalies and prevent repeat-offender developers from uploading applications on the Google Play Store [19]. Subsequent researches have proven that Bouncer is not effective enough, being easily bypassable and having a very limited rule-set, resulting in a low detection rate [10, 13]. Google have enhanced their defence mechanism with Verify Apps in July 2013, for malware detection at installation time, and Safety Net in February 2014, for detection of malicious behaviours during execution. However these tools are tied to Android updates and can be manually disabled by the user.

DroydSeuss is an Android malware tracker developed by Coletta et al. in October 2014, which performs a static analysis of the application code and executes it in a sandbox, tracks the network activity in order to retrieve information about the connections performed, according to the most popular protocols (e.g., HTTP, phone number, SMS, Google Cloud Messaging, . . .).

The threat agents must communicate with an endpoint in their control, a *Command and Control server* (C&C server), from which the malware receives instructions and to which it sends the stolen information; a C&C server is usually a web domain or a telephone number.

In this work, we present an extension to DroydSeuss that allows the security analysts to perform accurate searches on the apps previously scanned and find correlations among them, making them able to group malicious applications into clusters of similarity. We consider two apps to have a correlation if they share at least one C&C endpoint or have a similarity in the package name. In order to be able to perform this evaluation and have a reliable result, we first had to enrich the data extracted from the applications' source code. We employed Androguard [4] to extract its components (i.e., the activities, services, broadcast receivers and content providers which the application is composed of) and other significant information that was not previously taken into account (e.g., the strings), shifting the analysis performed from only taking into account the endpoints to considering all the elements of the application. We classify the C&C servers according to the results of the examination offered by VirusTotal, a free online service that examines the resource pointed by URLs and files with several antivirus engines to detect malicious content [7], telling benign and malicious URLs or domains apart. A further analysis on the strings allows us to understand which are the markets targeted by the application. All the extracted information is then stored on ElasticSearch, a powerful full-text search engine [1]. On the web application developed for these features to be available, the users can perform queries filtering out the fields (i.e., domains, activities, services, receivers, strings, classes, permissions) they are not interested in and afterwards navigate the individual reports of the matching applications.

In order to find correlated apps, we compare the suspicious and malicious C&C servers in order to find at least one og them in common, meaning that they are under control of the same threat agent; we also investigate the package name of the selected one with all the other package names and consider them as related if they are similar, by having a small Levenshtein distance [11], which is the minimum number of single-character edits (i.e., insertions, deletions or substitutions) required to change one word into the other. If a developer follows the official guidelines regarding package naming conventions, approximately 50 to 65% of the packages across its applications will be constant. Consequently, a small Levenshtein distance implies two samples have been developed by the same company or have been automatically generated by the same malware producer. Those applications that fulfil at least one of these requirements are then listed as apps that have a correlation and can be easily grouped as being part of the same malicious

campaign.

To summarize, we present the following contributions.

- We add an advanced search feature to DroydSeuss to perform queries on the already analysed samples.
- We introduce an extension to DroydSeuss to find correlations between applications and clusters of similarity.
- We provide an empirical benchmark of the features showing that the results are coherent with the goal set and that this work is scalable to the fully working DroydSeuss website.

This document is further outlined as follows. In Chapter 2, we examine the history of malware on Android, the attempts made by Google to stop its spreading and how DroydSeuss is meant to prevent banking trojans. In Chapter 3, we present the third party tools and services we used in our work. In Chapter 4, we discuss the implementation notes along with the design choices we made. In Chapter 5, we take a close look at the performance and scalability of our work, we present the scenarios we analysed and discuss the limitations of our work. Finally, in Chapter 6 we propose a number of possible future directions of research and possible extensions to DroydSeuss, then conclude our work.

Chapter 2

Background and motivations

Before we discuss the details of our work, it is necessary to understand what created the need for smart Android malware detectors. We examine the evolution of malicious applications and the effort spent by Google to contrast them in Section 2.1. We introduce DroydSeuss in Section 2.2, focusing our attention on the integrated components and the results achieved.

2.1 Malware growth in Android

We begin this brief historical examination with a breakdown of the reasons that brought Android to the powerful market position it has nowadays in Section 2.1.1; we present the work done by Google to fight malicious applications in Section 2.1.2; we conclude in Section 2.1.3 with an insight on banking trojans, the main threat under DroydSeuss analysis.

2.1.1 Android market share expansion

Since the first public release in October 2008, Google decided to adopt an open philosophy regarding the access to the operating system code, which is open source, and the availability of apps. Although the official distribution channel is the Google Play Store[1] (formerly Android Market), a user can install applications downloaded both from third-party stores, of which the most popular ones are Amazon App-Shop[2], GetJar[3] and SlideME[4], but also from unknown sources, such as Internet or compiled from the computer.

[1] https://play.google.com/store/apps

[2] https://www.amazon.it/gp/mas/get/androidapp/ref=mas_rw_gts

[3] http://www.getjar.com/

[4] http://slideme.org/

After few years struggling to gain a significant market share, since Q1 2011 Android has been the most worldwide widespread mobile operating system on smartphones and since mid 2013, period in which it passed 75% market share quota, also on tablets. This increase tendency is not going to stop in the next five years either: a compound annual growth rate of 6.9% will make Google able to reach the quota of over 1.62 billion devices shipped in 2020 [22, 9]. In the same period of time, the Google Play Store mirrored this high-speed growth, reaching 1 million applications released and over 50 billions applications downloads in Q2 2013, thus becoming the most popular app store across all platforms in July 2015, surpassing 1.6 million applications available. The leadership is going to become stronger, expecting an increase of 19.53% in worldwide downloads in 2017 [21, 20]. It is supposed that across all the third-party app stores, which in some countries are more diffused than the official one (e.g., in China), there are more than 1 million applications available, of which 20.000 are newly released every month.

2.1.2 Google's effort to prevent malware

In addition to having unregulated app stores, another potential threat is represented by the possibility for anyone to sign an Android applicationn: a cyber criminal can download a legitimate app, decompile it, insert malicious code, re-sign it with a new certificate (which can be created with no prior security check) and release it as if it were a copy of the original authentic application. Cyber criminals immediately recognized the opportunities offered by the exploitation of Google's loose control over their platform and, as soon as Android became dominant on the market, they began a meticulous development of malicious applications; in the timespan from 2011 to 2013, an increase of 388% of malware on the Google Play Store was detected, while the number of applications rejected or removed dropped from 60% in 2011 to 23% in 2013 [17]. In February 2012, Google reacted to these attacks releasing a service called Bouncer, which provides automated scanning of the Google Play Store for potentially malicious software, comparing the source code and functionalities with known malware, spyware and trojans. The goal of such a tool is to detect anomalies and prevent repeat-offender developers from having applications on the official channel[5]. Subsequent studies proved this tool not to be effective enough, having a limited rule-set, being easily bypassable and requiring manual updates to reflect the newly discovered vulnerabilities, in addition to being active only on the Google Play Store [16, 14]. Additional effort to prevent malicious applications on the devices has been employed in

[5] http://googlemobile.blogspot.it/2012/02/android-and-security.html

the following years, resulting in the release in July 2013 of a Potentially Harmful Applications (*PHAs*) detector at installation time, called VERIFY APPS, and a malware detector during the execution of the application, called SAFETY NET, in February 2014. Both these tools have demonstrated their efficacy against known ransomware campaigns, but they are tied to Android updates, can be manually disabled by the user and are still missing advanced studies to definitely prove their effectivness. [18].

2.1.3 Banking trojans

In this continuously evolving scenario, banking trojans, called *bankers* in jargon, have benefited from the spread of mobile devices as second factor of authentication [15]. A banker is a malicious program used to obtain confidential information about customers and clients using online banking and payment systems. Adapting famous crimeware kits, such as ZeuS[6], Torpig and SpyEye, to the current mobile scene, threat agents are able to perform man-in-the-middle (*MitM*) attacks, while the users are connecting to their online bank account, thus stealing one-time passwords and personal sensitive data. This kind of malware is particularly interesting for criminals and threatening for users because they allow for a great revenue in a small number of attacks. To fight these continuously evolving and improving threats in a concrete way, it is not possible to rely only on manual analysis and predefined triggers.

2.2 DroydSeuss design and achievements

DROYDSEUSS is a malware tracker based on reverse engineering designed to offer the security analyst an automated tool to inspect applications, avoiding the necessity of a manual inspection of the source code. It exploits the idea that a malware must communicate with a Command and Control, *C&C*, endpoint (e.g., a phone number, a server domain) in order to receive instructions and send the stolen data, obtained during the MitM attack, to the threat agent. These pieces of information are visible at runtime in the source code and/or in the bytecode, depending on the sophistication of the software. The finding of the endpoints both in static and dynamic analysis is considered an indication of being generated by a known crimeware kit. We explain in Section 2.2.1 how the data are extracted, then in Section 2.2.2 the operations done to give them greater significance, subsequently in Section 2.2.3

[6]https://zeustracker.abuse.ch/

how they are stored and finally in Section 2.2.4 the results obtained since the web application has been operative.

2.2.1 Data extraction

DroydSeuss decompiles the application `apk` file using TraceDroid, a scalable and automated framework for dynamic analysis of Android apps, based on Androguard[7] and baksmali[8] [23]. After having unpacked the app with `apktool`[9], the XML files and Smali bytecode are inspected in order to find web based endpoints (e.g., URLs, IPs and the push communication offered by Google Cloud Messaging, *GCM*) and phone based ones. After this static analysis, TraceDroid is employed to stimulate behaviours that are typically related to endpoint connectivity: sending of a text message (via `SmsManager`); connection to a web server (through `URL.openConnection` and `apache.*`); opening of a communication channel with GCM (using `GoogleCloudMessaging.*`). UI events, touches, gestures and click are generated in a pseudo-random fashion by the Android UI Exerciser Monkey[10]. The data extracted from the static analysis includes the following values.

package The unique identifier name given to application, written with the naming convention used in Java[11]

numbers The phone numbers retrieved from the application

endpoints The C&C servers retrieved from the application

significant The significant strings retrieved from the application

suspicious The suspicious strings retrieved from the application

use_GCM A boolean value to identify whether the application uses Google Cloud Messaging or not

GCM_ID The identifier of the sender in the Google Cloud Messaging connection

minSDK The minimum version of Android supported by the application

[7]https://github.com/androguard/androguard
[8]https://github.com/JesusFreke/smali
[9]http://ibotpeaches.github.io/Apktool/
[10]http://developer.android.com/tools/help/monkey.html
[11]http://www.oracle.com/technetwork/java/codeconventions-135099.html

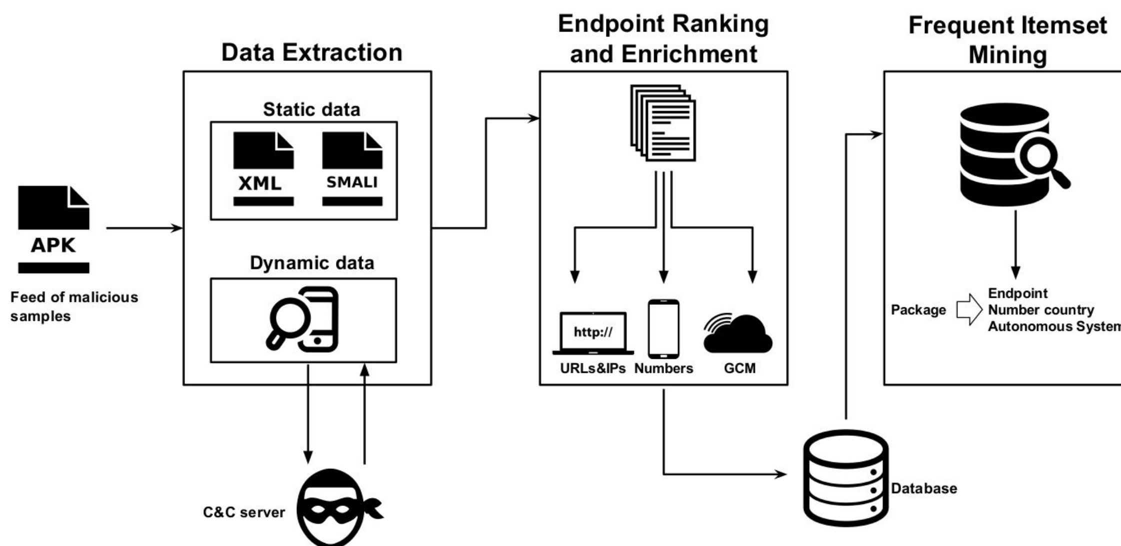


Figure 2.1: Data processing pipeline of DroydSeuss

In addition to the data obtained from the analysis, the following data are added.

sha256 The unique identifier value of the application we used to store and retrieve it from the database, obtained applying the SHA-256 cryptographic hash function the `apk` file under analysis

2.2.2 Data enrichment

After having obtained the endpoints information, it is enriched with additional information. IPs are localized using HostIP[12] and associated with their respective autonomous system using the service offered by Team Cymru[13]; phone numbers are localized through the Python library `pycountry`[14] and their type (e.g., tool free, mobile, landline, etc...) is extracted using a Python port of `libphonenumber`[15]. The endpoints are also classified in three rankings, according to various heuristics.

[12]http://www.hostip.info
[13]http://www.team-cymru.org/IP-ASN-mapping.html
[14]https://pypi.python.org/pypi/pycountry
[15]https://github.com/daviddrysdale/python-phonenumbers

Suspicious An endpoint found only in static analysis

Significant An endpoint discovered during dynamic analysis, but only in secondary functionalities

Important An endpoint matched during dynamic analysis and the active usage of an API function is detected, meaning that the malware actually used it

The phone numbers are in a similar fashion categorized in two groups.

num_end Endpoint numbers retrieved from the application

num_suspicious Suspicious numbers retrieved from the application

2.2.3 Data storage

The data are first stored in a temporary database table as they are collected and after the completion of all analysis operations, they are saved in the final database table. The system is based onMongoDB[16], a NoSQL document-oriented database, which allows to retrieve results in a schema similar to JSON, called Binary JSON or *BSON*[17]. A BSON document is an object composed by an ordered list of elements, which consists of a string, used as field name, a type and a value. Using BSON makes storage and scan operations more efficient than traditional databases: large elements in a BSON document are prefixed with a length field to facilitate scanning, although this solution might lead to more space occupied. The load balance is optimized by horizontal scaling through sharding; this technique allows to add more machines to support data growth and the demands of read and write operations.

The database used by the first version of DroydSeuss relies on a table with the document schema detailed in Table 2.1; the information about the content of each key field can be found in the bullet lists in Section 2.2.1 and Section 2.2.2.

2.2.4 DroydSeuss today

In this section, we first have a look at what is the first generation of DroydSeuss and its achievements, then we introduce the innovations we are proposing.

[16]https://www.mongodb.org/
[17]http://bsonspec.org/

Key	Type	Additional information
sha256	`string`	primary key
package	`string`	not null
min_sdk	`int`	
gcm	`boolean`	
gcm_id	`string`	
number_endpoint	list[`string`]	
number_suspicious	list[`string`]	
endpoints	list[`string`]	
significant	list[`string`]	
suspicious	list[`string`]	

Table 2.1: DroydSeuss original database schema

Current state

The final work consists in a web application in which the user can submit a sample to analyse and, after about 6 minutes of examination, receive a report listing all the threats found. The data in Table 2.2 show an example of a report with all data fields filled with values. After being publicly released in October 2014[18], DroydSeuss received 1187 samples over the following 4 months. It produced a blacklist of the extracted endpoints and revealed a campaign active at that moment against Chinese and Korean bank customers. It is possible to have an insight over aggregated statistical data about the localization of the C&C servers; information about the type and country of the phone numbers; the recurrence of certain package prefixes; the recurrence of second-level domains. Lastly, it is possible to download the blacklists produced about the malicious IP addresses, domain names and endpoint numbers.

Proposed extensions

In order to offer a better automation in malware recognition, we propose two major contributions to the current state of the tool. The implementation details about them are listed in Chapter 4.

- We open our database to the public with a smart search engine. We think that, if the applications can be stored and retrieved only by the person who submitted it, this may not prove enough to match the ever growing harmfulness of malware. By allowing the users to search for

[18] http://ds.andrototal.org/

Endpoint numbers	+44 7781 470730 - Guernsey (GG) - Mobile
Suspicious numbers	+7 906 707-51-45 - Russian Federation (RU) - Mobile
C&C servers	myredskins.net (67.215.65.133, None, BRAZIL (BR), -23.5666, -46.6333) mynamesmith.com (67.215.65.133, None, BRAZIL (BR), -23.5666, -46.6333)
Significant strings	http://myredskins.net/iBanking/sms/index.php http://mynamesmith.com/iBanking/sms/ping.php http://myredskins.net/iBanking/sms/ping.php
Suspicious strings	http://tmn.pt/update/

Table 2.2: DroydSeuss report example

keywords across all stored samples and in all the content they carry (not only the malicious one) we offer the opportunity to easily find applications which have already been analysed by other users and read the report about them.

- We introduce the possibility of finding correlations between applications, grouping them in clusters of similarity. We know that there are several crimeware kits available on the market, as we explained in Section 2.1; the apps which are developed in such a way inherit some common characteristics that allow us to backtrack to their common source. This knowledge permits us to predict the maliciousness of an application by comparing its contents to the ones of others previously examined known to generate suspicious behaviour, thus classifying it without having to perform a detailed analysis.

Chapter 3

Architecture

In this chapter, we are presenting the third-party components we used, how and why we integrated them in our work and how the data flow streams across the web application. In Section 3.1, we present the third party components and tools needed for our work. In Section 3.2, we explain how they communicate and exchange information with each other.

3.1 Components

In Section 3.1.1, we talk about the already implemented third party components we used to achieve our goals, why we needed them and what we used them for; in Section 3.1.2, we explain which needs couldn't be satisfied by the original architecture and thus introduce the new tools we needed to work with.

3.1.1 Legacy components

In this section we will talk about the tools already incorporated in the original version of DroydSeuss and how we employed them to fulfil our needs.

Androguard

Androguard is a full Python tool capable of disassembling and decompiling the files that compose an Android application, extracting knowledge and design information and analysing its components and workings in detail [4]. It was already employed to extract part of the static data stored in the XML files of the application, as we discussed in Section 2.2.1. We exploited its functionalities to extract other components (i.e., the activities, services, broadcast receivers and content providers which the application is composed

From .apk disassembler	From .dex disassembler
Package	Classes
Android SDK version	Strings
Application name	Methods
Activities	Fields
Services	
Receivers	
Permissions	
Permissions details	

Table 3.1: The source of the data we employed

of) and other significant information that was not previously taken into account (e.g., the strings), shifting the analysis performed from only taking into account the endpoints to considering all the elements of the application. The details about the extension of the data extraction are examined in Section 4.1.1.

In order to achieve the goals we set, we needed to use more functionalities offered by ANDROGUARD: in the original version of DROYDSEUSS, only the .apk file was taken into account and not all the available data were taken; we extracted more information from the .apk file and also analysed the .dex file. Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. The greatest utility .dex files have for us is holding a set of class definitions and their associated adjunct data. In Table 3.1 we summarize the source from which we got the data we have worked on.

3.1.2 New components

In order to fulfil the requirements originated by the new features we wanted to add, we needed tools capable of fitting our needs.

VirusTotal

VIRUSTOTAL is a free online web application that aggregates several antivirus products and online scan engines, in order to verify if a file or a

webpage is malicious or not [7]. It was already used in the original version of DroydSeuss but only to test the URLs extracted; since we wanted to broaden our recognition power, we needed to inspect the domains or IP addresses. We started from scratch as if it was a completely new service, writing our communication methods and setting the thresholds we wanted. Further information about the integration with this tool is detailed in Section 4.1.1.

Elasticsearch

Since we wanted to allow the users to perform queries on our database, we needed to move it from being placed locally to being located online. Elasticsearch[1] is a search server that provides multitenancy, scalable searches and near real-time finding of information. It is distributed in shards, which can be replicated, divided in nodes; this architecture allows to have operations delegation, rebalancing and routing automatically handled by the system. The communication interface used is JSON for both the documents storage and the definition of the schema, which can be automatically created from the detection of the document structure and types or can be manually customized [1]. We outline the steps we followed to integrate this tool in Section 4.2.

3.2 Communication flow

We present an overlook to the architecture components and how they communicated with each other. In Section 3.2.1, we introduce the data flow of the original version of DroydSeuss and the additions we had to make to it. In Section 3.2.2, we present how to perform a search on the database and in Section 3.2.3 we details the steps needed to find applications similar to a given one. The details about the implementation of these features are listed in Chapter 4.

3.2.1 Structure of the analysis

The goal of the first DroydSeuss version was to analyse applications and find malicious content. In order to achieve this, the user selects the `.apk` file to be analysed and submits it. While the client handles the request, a new web page is shown to the user, informing that the time needed for the analysis to be performed is at least 6 minutes. The back end first computes the SHA-256 value of the application and checks if it has already been analysed;

[1] https://www.elastic.co/products/elasticsearch

if this is the case, the stored data are retrieved, otherwise the analysis is initiated. TraceDroid is started to parse the XML files and the extracted data are saved in a temporary database. Then, the smali files are analysed; the extracted data, together with the stored ones, are enriched by consulting VirusTotal service and are finally stored in the database. In order to fulfil the user request, the sample's information is retrieved from the database, sent to the client, which updates the waiting page and redirects the user to the analysis report. A schema of this data flow is illustrated at Figure 3.1.

To accommodate the needs of the expansion due to our work, we needed more data, a more refined data enrichment and the storage on an online distributed database. For the sake of data flow, we needed to add Elasticsearch to our architecture and send the data to it, after the completion of the analysis, as exemplified in Figure 3.2.

3.2.2 Perform a search

When executing a search, the user first types the keywords he wants to query and presses the *submit* button. While the search is initiated, a progress bar is shown to the user to make him aware of the necessity to wait. The client forwards the request to the back end, which establishes a connection with the online database; after a JSON reply is received, the structure is parsed to extract the needed information. The data are sent to the client, which updates the search page in order to display the results retrieved. A schema of these interactions is displayed in Figure 3.3.

3.2.3 Find similar applications

After having analysed an application, the user can find other samples correlated to it. The request is started by clicking the specific button: the back end is informed of the task and the client shows a progress bar to inform the user of the necessity to wait. The server connects to Elasticsearch in order to retrieve all the stored samples, extracts the chosen app from the full list and then tries to find relationships between them. After the computation, the data are sent to the client, which redirects the user to a page where the report about the similar applications found is shown. Figure 3.4 shows a sequence diagram of this communication path.

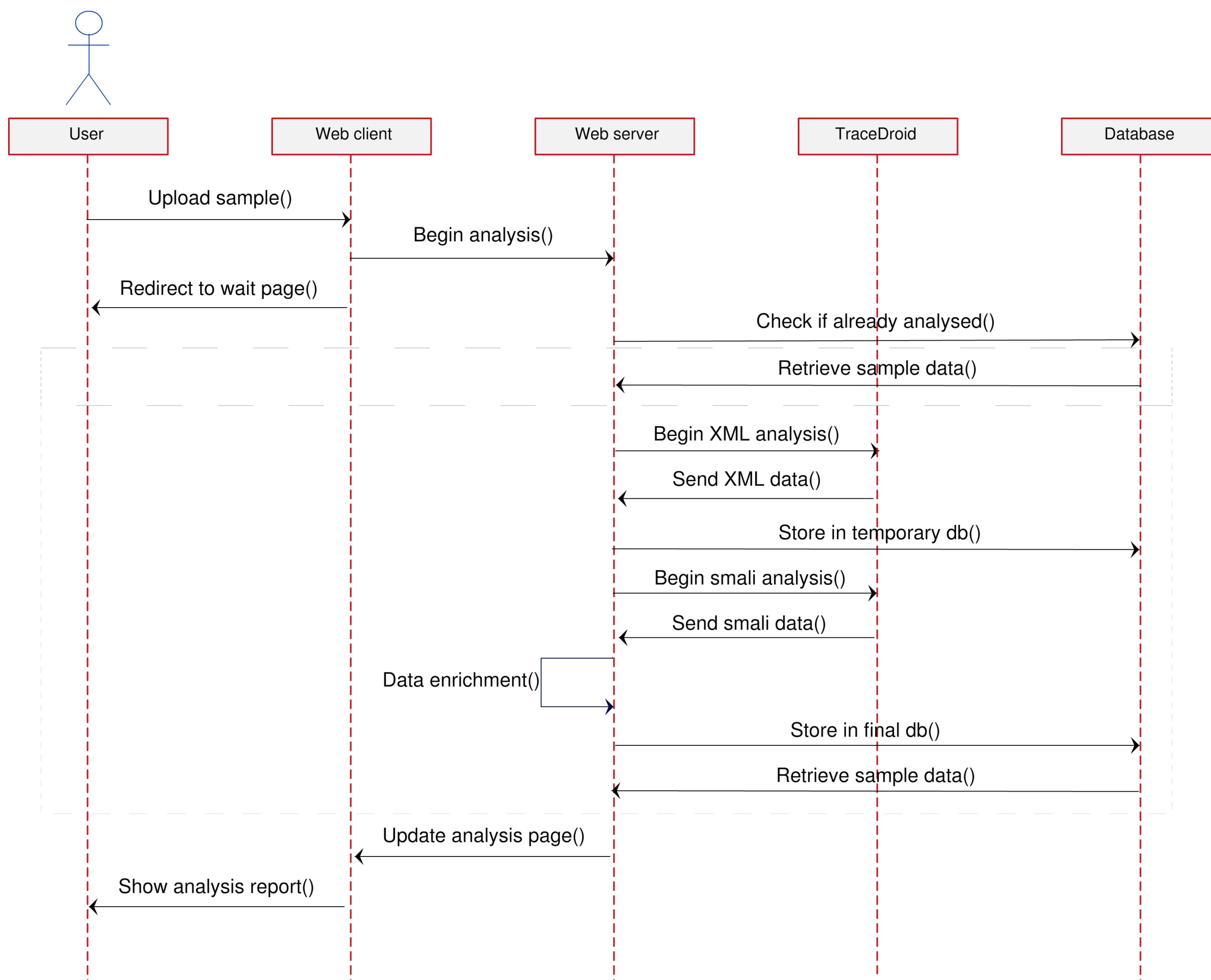


Figure 3.1: Original DroydSeuss analysis sequence

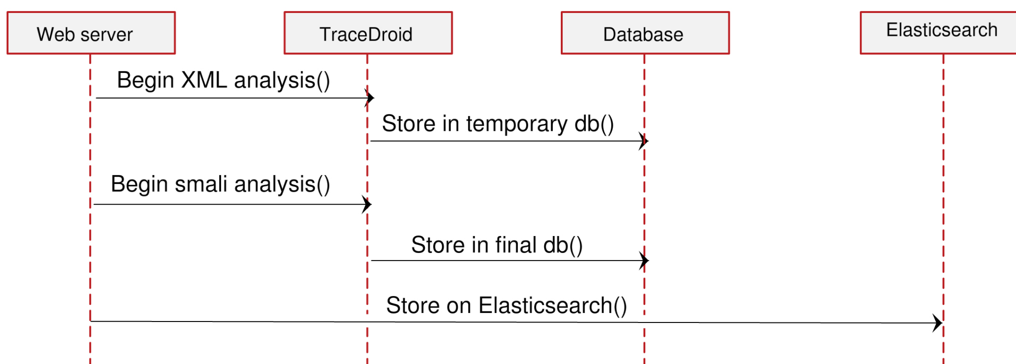


Figure 3.2: Addition to DroydSeuss analysis sequence

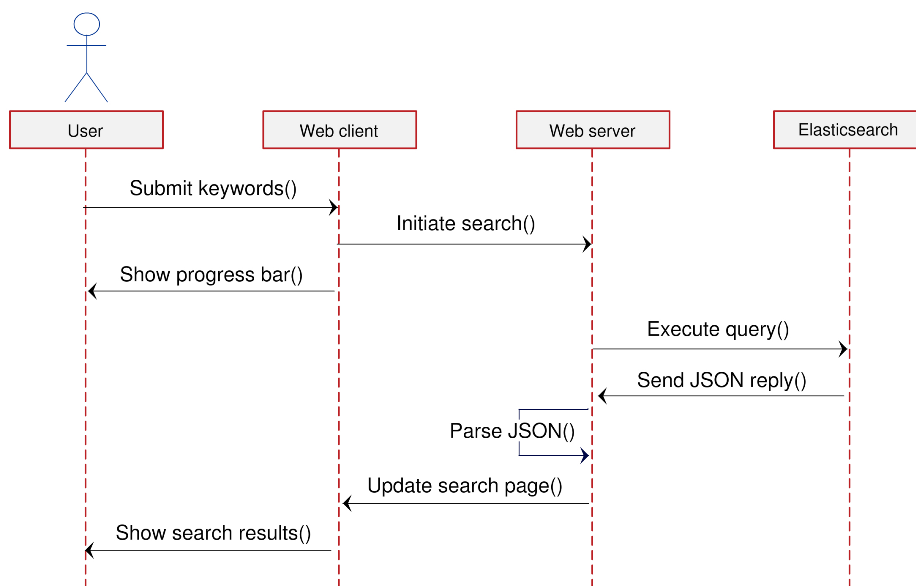


Figure 3.3: Data flow in a search

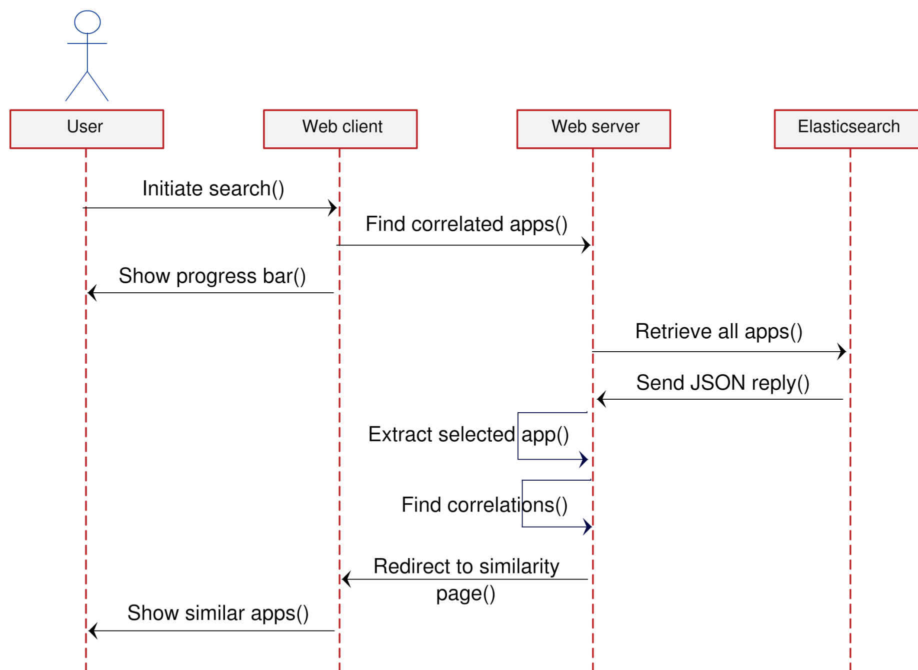


Figure 3.4: Data flow in a research of clusters of similarity

Chapter 4

Implementation

In this chapter, we first describe how we improved the static analysis of Android applications in Section 4.1; we discuss how we moved our database from a MongoFB database to a distributed multitenant one in Section 4.2; we describe the algorithms we implemented in order to find clusters of similarity in Section 4.3; we conclude presenting the web pages related to our work in Section 4.4.

4.1 Static analysis improvement

We describe in Section 4.1.1 the additional pieces of information we needed and in Section 4.1.2 the implementation of a deeper and more refined static data analysis, based on Androguard disassemble and decompile tools [4].

4.1.1 Extension of the extracted data

The first implementation of DroydSeuss offered a limited analysis of the application, in terms of quantity of data, because it focused solely on the C&C servers. We extended the analysis to other resources that can be found in the XML structures of an Android application.

Application name The name of application, shown to the user in the device launcher

Permissions The permissions, and the details about their functionality, required by the application to perform actions that would impact the user experience or any data on the device

Activities The application components that provide a screen with which the user can interact and the tree structure in which they are located

Services The application components that can perform long-running operations in the background and do not provide a user interface

Receivers The component that enables applications to receive intents that are broadcast by the system or by other applications, even when other components of the application are not running

Classes The templates that are used to create objects, and to define object data types and method

Strings The character sequences contained in the application

4.1.2 Data enhancement

In this section we detail the procedures we followed in order to give a higher significance to the data we were working on.

Domains extraction

The C&C servers pieces of information we got from the application XML are normally related to sub-pages, files, contain details about the firewall port or are non consistently structured. In general, it adheres to the pattern `protocol://(www)?.(subdomain)*.domain.extension(:firewall_port)?(/subpages)*` or the pattern `ip_address(:firewall_port)?`. In order to have significant data to work with, we needed to extract only the domain from the full URL and we did it by using the regexen described at Listing B.1.

Additional domains recognition

A manual inspection revealed that some domains and subdomains, particularly those related to advertising or part of important corporations, may lead to misleading results during the analysis.

- A malicious service can advertise on a benign platform. Suppose a harmless website called `http://www.harmless_ads.com` hosting a campaign paid by *malicious_malware* will generate a domain `http://www.harmless_ads.com/campaign?=malicious_malware` and thus possibly categorizing the innocuous website as malicious

- A malicious domain may have in its URL structure the name of a benign company or service. Suppose a malicious endpoint called `http://www.data_stealer.com` which hosts in its subdomain called `java`

the related code written in that language. The final URL `http://www.data_stealer.com/java/`, in addition to other instances of the same recurring keyword, may lead to supposing that `java` is related to malevolent attacks

Consequently we excluded the domains `schemas.android`, `admob.`, `java.` and `android.clients.google.com` from the endpoints we got from the static analysis as being not interesting and possibly hinting to misleading results from the subsequent analysis.

IP standardization

After having filtered out the domains, we needed to analyse the IPs. We implemented in Python a version of the pseudocode shown at Algorithm 1. An IP is structured as four numbers divided by a dot, optionally followed by a colon and the port number used to communicate. We needed to both remove the incorrect IPs and uniformly format the correct ones (e.g., for our purposes `93.1.216.34` is the same as `093.001.216.034`). In order to achieve this goal, we used the regex presented at Listing B.2 and followed the steps listed hereafter.

1. We removed the port number, if present
2. We split the components of the IP and discarded the ones that were not made by four of them
3. We discarded the IPs containing characters different from digits
4. We discarded the IPs containing numbers not in the range $[0, 2^8 - 1]$, which are the only ones allowed by the IPv4 notation [5]

Endpoints maliciousness check

We consulted VirusTotal[1] service in order to have an insight over the maliciousness of the endpoints. We first send each domain to the `scan API` at `https://www.virustotal.com/vtapi/v2/url/scan` and afterwards retrieve the result from the `report API` at `https://www.virustotal.com/vtapi/v2/url/report`.

As of April 2016, VirusTotal interrogates 67 URL scanners. In order to prevent false positives due to few antiviruses reports, we decided to have three

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

Algorithm 1 Pseudo code for IP standardization

```
 1: function StandardizeIP(IP_in)
 2:     b = IP_in.split(':')
 3:     a = b[0].split('.')
 4:     IP_out = ''
 5:     if a.length() ≠ 4 then
 6:         return null
 7:     for x in a do
 8:         if x is not a digit then
 9:             return null
10:         i = integer(x)
11:         if i < 0 or i > 255 then
12:             return null
13:         IP_out ⊔ '.' + i
14:     return IP_out
```

different ranks of maliciousness, called *clear*, *maybe* and *malicious*. We set a threshold equal to $\lfloor 3\% \rfloor$ of the total number of antiviruses, to discriminate between the domains known to be surely malicious and the ones that are malicious only for a little number of detectors - the ones we put in the category *maybe*.

Localization detection

Having access to the strings contained in the application, we decided to analyse them in order to understand which are the targeted geographical areas of each of them. Employing the Unicode Character Ranges, described in Table 4.1, we searched for characters belonging to the following categories: Latin characters, Chinese, Japanese and Korean (*CJK*) characters, Cyrillic characters, Greek characters and Arabic characters. Then, we stored the information if the application contains a certain kind of characters.

Ranges	From	To	Explanation
Latin	0000	007F	Latin
	0080	00FF	Latin-1 Supplement
	0100	017F	Latin Extended-A
	0180	024F	Latin Extended-B
	1E00	1EFF	Latin Extended-Additional
CJK	03A0	30FF	Japanese Kana
	2E80	2EFF	CJK Radicals Supplement
	3300	33FF	CJK Compatibility Ideographs
	3400	4DBF	CJK Compatibility Ideographs Extension A
	4E00	9FFF	CJK Unified Ideographs
	F900	FAFF	CJK Compatibility Ideographs
	FE30	FE4F	CJK Compatibility Ideographs
	20000	2A6DF	CJK Compatibility Ideographs Extension B
	2A700	2B73F	CJK Compatibility Ideographs
	2B740	2B81F	CJK Compatibility Ideographs
	2B820	2CEAF	Included as of Unicode 8.0
	2F800	2FA1F	CJK Compatibility Ideographs Supplement
Cyrillic	0400	04FF	Cyrillic
	0500	052F	Cyrillic Supplement
	2C00	2C5F	Glagolitic
Greek	0370	03FF	Greek
	1F00	1FFF	Greek Extended
Arabic	0600	06FF	Arabic
	FB50	FC3F	Arabic Presentation Forms A
	FE70	FEFF	Arabic Presentation Forms B

Table 4.1: Unicode Character Ranges

4.2 Data persistence and retrieval

In Section 4.2.1, we first describe the tools selected for the integration with ELASTICSEARCH and the mapping chosen. In Section 4.2.2, we detail the data structure we employed

4.2.1 Libraries integration

Since DROYDSEUSS back end is Python based, we decided to use both ELASTICSEARCH official clients for that language: the low-level one, called `elasticsearch-py`[2], and the high-level one, called `elasticsearch-dsl`[3].

In order not to lose coherence in our data, due to ELASTICSEARCH indexing, we had to manually set the mapping of our online documents.

- We decided to have few `boolean` fields mapped to `string`, to have greater flexibility when we had to work on them after we retrieved them.
- Due to the methodology of ELASTICSEARCH, two strings separated by a dot or a dash (e.g., *cogito.ergo-sum*) are stored as separated values (e.g., *cogito*, *ergo*, *sum*); as a consequence, when searching for the the full concatenation of word, no result is retrieved. Since we needed to be able to search for values containing the aforementioned special characters (e.g., URLs, packages, etc. . .), we mapped the related fields at both `string` and `not_analyzed`, to be able to query both the full stored value and part of it.

We created a mapping style at `_index` "droid-1" and `_type` "app" which contains all the information extracted from the application, as it can be seen in Listing A.1. After completing this step, when we wanted to add an application to the online database or update an existing one, we could index the document using the `_index` and `_type` defined before, employing the application `sha256` as `_id` and the extracted information as `_body`.

`elasticsearch-py`

This low-level client offers easy to use APIs to establish the connection and post data to the server; the search APIs, however, is very verbose, prone to syntax mistakes and hard to modify. For this reason, we used this library only for the first two purposes. Examples of its usage are listed in Section 4.2.3

[2]https://elasticsearch-py.readthedocs.org/
[3]https://elasticsearch-dsl.readthedocs.org/

`elasticsearch-dsl`

We exploited the conciseness of this higher-level client to write more maintainable and easier to read queries to retrieve the data from ELASTICSEARCH; this library offers chainable methods over objects, access to response data and avoids the use of nested brackets. We created procedure to retrieve a stored app, given its `sha256` unique identifier; a procedure to fetch all stored apps, filtering out the not needed fields; a procedure to search an unlimited number of keywords in a user-defined amount of fields; a procedure to find the common elements among the fields of two apps; a procedure to find the Levenshtein distance [11] between a given app's package name and those of all the other ones stored. Examples of the aforementioned usages are provided in Section 4.2.3

4.2.2 Data structure

In order to provide complete retro-compatibility, we had to have a structure on the online database coherent with respect to the schema already present on the local database, described in Table 2.1. Thanks to ELASTICSEARCH flexibility, we could translate the data originally of type `list[*]` to a nested JSON dictionary with the field as `key` and the content of the list as `values` of type `*`. An exemplification of this transformation can be seen in Table 4.2. We modelled the new data according to the type we set when defining the mapping of our index (see Listing A.1) and thus resulting in a data structure as explained in Listing 4.1.

4.2.3 Elasticsearch usage

In this section, we detail some examples of usage of ELASTICSEARCH REST APIs and of the Python libraries presented in Section 4.2.1. Thanks to `elasticsearch-py`, we could easily establish the connection by importing the related declaration and invoking it at the beginning of our code.

Sample storage

After the application's analysis, we could store the data extracted as shown in Listing 4.3, without having to manually define the whole JSON structure of the request.

<table>
<tr><th>Name</th><th></th><th></th></tr>
<tr><td><code>list[</code> "Ann" "Bob" "Charlie" <code>]</code></td><td>→</td><td><pre>"name": {
        "Ann",
        "Bob",
        "Charlie"
        }</pre></td></tr>
</table>

Table 4.2: Data translation from MongoDB to Elasticsearch

Single sample retrieval

Since we decided to assign the `sha256` value as `_id`, we could easily retrieve each application by sending an empty body request, as exemplified in Listing 4.2. However, we preferred to use the methods offered by the libraries, in order to keep uniformity across our code, as shown in Listing 4.4 at line 7.

All samples retrieval

When searching for Levenshtein similarity and elements in common between applications, we executed a query with no parameters (in order to match all samples), potentially filtering out the fields we did not need. Since the connection time to Elasticsearch was the most time consuming, as we detail in Section 5.1, it was essential not to fetch unneeded values. The two related pieces of code are both presented in Listing 4.4 at lines 12 and 15.

Advanced search with parameters

We needed a different query for when the user performs a search from the search bar, because we wanted to filter the database according to the input keywords. At line 18 in Listing 4.4 it is shown the query we employed: it's a `multi-match` one, in order to retrieve more than one result, with the keywords used as parameters and the checkboxes selected (as explained in Section 4.4.2) used as fields; then, we returned only `name` and `package`, since we didn't need the other values.

Listing 4.1: Elasticsearch data structure

```
"GCM_ID": {
  "type": "string"
},
"activities": {
  "type": "string",
},
"classes": {
  "type": "string",
},
"endpoints": {
  "type": "string",
},
"hasArabic": {
  "type": "boolean"
},
"hasCJK": {
  "type": "boolean"
},
"hasCyrillic": {
  "type": "boolean"
},
"hasGreek": {
  "type": "boolean"
},
"hasLatin": {
  "type": "boolean"
},
"ips": {
  "type": "string"
},
"malicious_domains": {
  "type": "string",
},
"minSDK": {
  "type": "integer"
},
"name": {
  "type": "string"
},
"num_end": {
  "type": "string"
},
"num_suspicious": {
  "type": "string"
},
"numbers": {
  "type": "string"
},
"package": {
  "type": "string",
},
"permissions": {
  "type": "string",
},
"receivers": {
  "type": "string",
},
"safe_domains": {
  "type": "string",
},
"services": {
  "type": "string",
},
"sha256": {
  "type": "string"
},
"significant": {
 "type": "string",
},
"strings": {
  "type": "string",
},
"suspicious": {
  "type": "string",
},
"suspicious_domains": {
  "type": "string",
},
"use_GCM": {
  "type": "string"
}
```

Listing 4.2: Elasticsearch GET API

```
curl -XGET 'http://localhost:9200/droid-1/app/<sha256>'
```

Listing 4.3: `elasticsearch-py` usage

```python
from elasticsearch import Elasticsearch

es = Elasticsearch()

# Sample storage
es.index(index='droid-1', doc_type='app', id=sha256, body=<app data>)
```

Listing 4.4: `elasticsearch-dsl` usage

```python
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search
from elasticsearch_dsl.query import MultiMatch

es = Elasticsearch()
s = Search(es)[0:10000]

# Retrieve a sample given its sha256 value
s.query('match', sha256=<sha256>).execute()

# Retrieve all samples
s.query().execute()

# Retrieve all samples, selecting only certain fields
s.fields(<fields list>).execute()

# Retrieve all samples matching the keywords entered
s.query('multi_match', query=<keywords>, fields=<fields list>)
 .fields(['name','package']).execute()
```

4.3 Clusters of similarity

We are presenting the details about the algorithms we used to find correlations between samples and the reasoning that brought us to choose these metrics: we discuss the concepts behind finding common endpoints in Section 4.3.1 and the ones behind package similarity in Section 4.3.2. We consider two or more applications that belong to a malware campaign as being part of the same *cluster of similarity*.

4.3.1 Common endpoints

We suppose two applications to be related if they share at least one malicious or suspicious endpoint. If this is the case, it implies they are controlled by the same threat agent, who can retrieve the stolen data from both apps.

In order to find common C&C servers, we select two applications: one is chosen by the user and the other one is selected from the pool of all the stored samples. We aggregate the suspicious and malicious endpoints for each of them and check if they share any of them. This process, as depicted by pseudocode in Algorithm 2, is repeated for all the applications stored on Elasticsearch.

Collection of other common data

After having positively checked the correlation between the two applications, we examine all the other pieces of information we have, in order to find common activities, permissions, services, receivers and string localization. All the common data found are presented to the user in a report, as shown in Section 4.4.2.

Algorithm 2 Pseudo code of similarity finding

1: **function** FindSimilarity($app_1[sha256]$)
2: $apps \leftarrow$ all samples from Elasticsearch
3: **for** $app_j \in apps$ **do**:
4: **if** $app_j[sha256] = app_1[sha256]$ **then**
5: Extract app_1 from $apps$
6: **for** $app_j \in apps$ **do**:
7: **if** $app_j[sha256] \neq app_1[sha256]$ **then**
8: $app_1[domains] \leftarrow app_1(suspicious \cup malicious\ domains)$
9: $app_j[domains] \leftarrow app_j(suspicious \cup malicious\ domains)$
10: $common \leftarrow$ common elements($app_j[domains], app_1[domains]$)
11: $distance \leftarrow$ levenshtein distance(app_1, app_j)
12: $threshold \leftarrow \min(app_1.size(), app_j.size())/2$
13: **if** $common.size() > common.size()/domains.size()$ **or** $distance \leq threshold$ **then**
14: Find other common elements between app_1 and app_j

4.3.2 Packages similarity

Many different approaches have been implemented during the recent years to detect correlations among Android apps [3, 12, 8]; since we are working on a real-time web application, we needed a responsive and lightweight method to recognize if two applications belong to the same developer, whether it's malicious or not. Exploiting the Android guidelines to avoid conflicts with other developers while naming packages[4], we decided to employ Levenshtein distance as a metric [11].

Levenshtein distance

Levenshtein distance is a string metric for measuring the difference between two sequences of characters, which is the minimum number of single-character edits (i.e., insertions, deletions or substitutions) required to change one word into the other. The mathematical definition can be seen at Formula 4.1, while the pseudocode representation of the implementation we used can be found in Algorithm 3. It differs from the Hamming distance because it supports strings of different length; in case the two strings are equally long, the Hamming distance is an upper bound on the Levenshtein distance [6].

As an example, given the words `app` and `laptop`, their Levenshtein distance equals 3, because there is no way to get from one word to the other with fewer manipulations.

$$
\begin{aligned}
&app \rightarrow \mathbf{l}app && \textsc{Distance} = 1\\
&lapp \rightarrow lap\mathbf{t}p && \textsc{Distance} = 2\\
&laptp \rightarrow lapt\mathbf{o}p && \textsc{Distance} = 3
\end{aligned}
$$

$$
\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \tag{4.1}
$$

Android package name structure

Google suggests to name packages, which have to be unique on the device and on the Google Play Store, using at least three identifiers, separated by dots, in order to avoid conflicts.

[4]http://developer.android.com/guide/topics/manifest/manifest-element.html

Algorithm 3 Pseudo code for calculation of Levenshtein distance

```
1: function LevenshteinDistance(String s, String t)
2:     m = s.length()
3:     n = t.length()
4:     d = matrix[m][n]
5:     for i in 0...m do:
6:         d[i][0] = i
7:     for j in 0...n do:
8:         d[0][j] = j
9:     for i in 0...m do:
10:        for j in 0...n do:
11:            if s[i] = t[j] then:
12:                d[i][j] = d[i-1][j-1]
13:            else
14:                d[i][j] = min(d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1]+1)
```

Localization suffix A two-three letters code that designates the localization of the company. It should be the Internet top-level domain of the company's website, if it has one.

Company name The name of the developing company. It should be the second level domain of the company's website, if it has one.

App identifier The name of the application or an internal place-holder used by the company to identify the application

As an example, Google's website is `http://www.google.com` and thus its applications packages are named `com.google.`*application_name*. If a company has subdivisions that work on different kinds of applications or an application has more than one version available, it's advisable to add more identifiers to the package name. If company *Example*, which owns website `http://www.example.com`, develops mobile games of different sports, it should name its packages `com.example.soccer.`*game1*, `com.example.soccer.`*game2*, `com.example.basketball.`*game3*, etc...

From manual analysis, we discovered that several automatically generated malware are named following this convention but present nonsensical strings as application identifier; moreover, they often differ for just a few characters. Keeping this in mind, we are able to take advantage of Levenshtein distance, because it will result in a low value, compared to the actual length of the package name. We estimated that, given the same localization

domain and the same company name, even having totally unrelated application names, the Levenshtein distance between the two given applications will be less or equal to half the length of the shortest package name, as shown in Formula 4.2. After having found a correlation through Levenshtein distance, the subsequent steps of analysis to finalize a report to show to the user are the same as described in Section 4.3.1.

In Section 5.2.1, we show a real case study of a malicious campaign discovered and in Section 5.2.3 we show how this technique allows to group benign applications of the same developer.

$$threshold = \frac{\min\left[package_1.\text{length}(), package_2.\text{length}()\right]}{2} \tag{4.2}$$

4.4 Web application

After having extended the functionalities of DroydSeuss, the challenge we faced was the development of an easy to use interface that shows only the most important parts of the analysis, in order to be a concrete help to the security analyst. We begin explaining the reasoning behind the design and user experience choices we made in Section 4.4.1. In Section 4.4.2, we present the look and feel of our client side work.

4.4.1 Design choices

We were inspired by Google's minimalistic design for their successful search engine. We decided to implement a `textbox` where the users can type in their query text and a button to start the search; in addition, we added a `checkbox` for each different field in which to search for the keywords (i.e., domains, activities, services, receivers, strings, classes and permissions). We make them aware of how much time is required to perform the search they request by displaying this information among with the results retrieved. In order to increase the user-friendliness of our work, we took the following design decisions.

- The checkboxes are all beforehand selected. The users must manually exclude a field from their search rather than select the fields they want to perform it with. The reasoning behind this choice is that we suppose that the users want at first to maximize the chance of finding matches to their keywords and perhaps refine their search afterwards, filtering out some results.

- If an empty search is started, instead of asking the user to type a keyword, we retrieve the entire database as if the user performed a `SELECT *` kind of query.
- The keywords are displayed in the search bar after the search has been completed, although they are not taken into account if the user wants to perform a new one.
- The applications are presented with their name and package, instead of their SHA-256 value. It's a less precise technique to identify a sample but it's much easier for the user to find again an app this way rather than having to remember a 64 alphanumeric character long string.
- We offered the choice to filter out the domains with only one checkbox. The users don't have to know the difference between safe, suspicious and malicious domains until they are reading the report, simplifying their experience. The inclusion or exclusion of all domain categorizes (*safe*, *suspicious* and *malicious*) is done with a black box approach, with respect to the user.
- We implemented a progress bar for both the search over the database and the search of clusters of similarity. Even though the time currently required is very small and is not expected to increase to too high levels, as we can see in Section 5.1.1 and Section 5.1.2, we took into account the possibility of connection issues or server congestion. By having a visual feedback from the page, the user knows that DroydSeuss is currently handling the request and can wait for its completion.

4.4.2 User interface and user experience

The entry point of the web application is the homepage. By selecting *Advanced search* in the menu at the top of the page (opposed to the old *Search by SHA-256* still present in top right corner), the page displayed in Figure 4.1 is reached. There, the user can find the search bar with the related filters. If the search retrieved at least one samples, a list of names and packages similar to the one in Figure 4.2 is shown, otherwise we inform the user that the search failed, as we can see in Figure 4.3. Clicking on the application name, the analysis report is displayed; we added a button to start the similarity search to the samples whose analysis extracted useful information, as shown in Figure 4.4. If the search for similarity does not find any correlated application, the page shown in Figure 4.5 is displayed and it's the end point of the path the user can follow; otherwise, a report similar to the one in Figure 4.6

listing all related samples is presented to the user. Only the relevant shared information are shown, including a coarse grain level of details about the characters localization and Levenshtein distance, if pertinent. By clicking on the application name, the user can directly go the see the details about the analysis report and from then on, continue a loop of retrieving clusters of similarity.

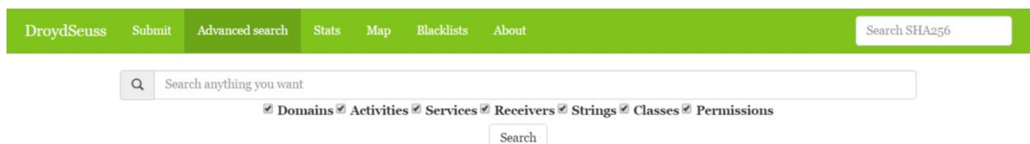


Figure 4.1: Advanced search feature entry point

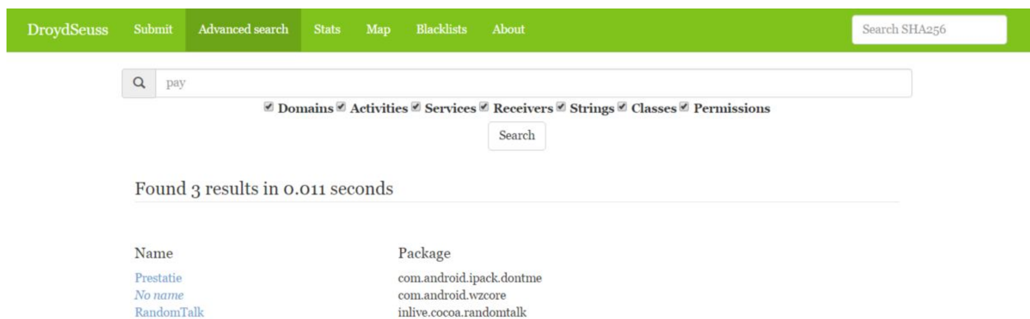


Figure 4.2: Search bar after a query with results found

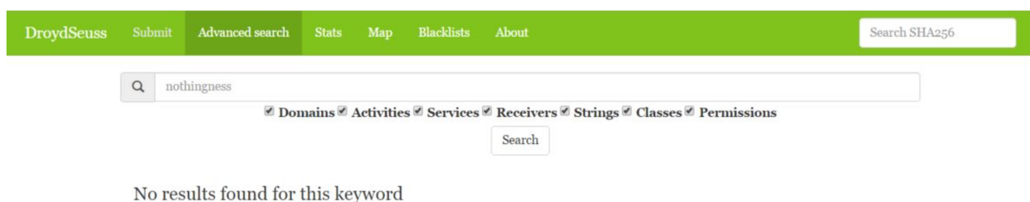


Figure 4.3: Search bar after a query with no results

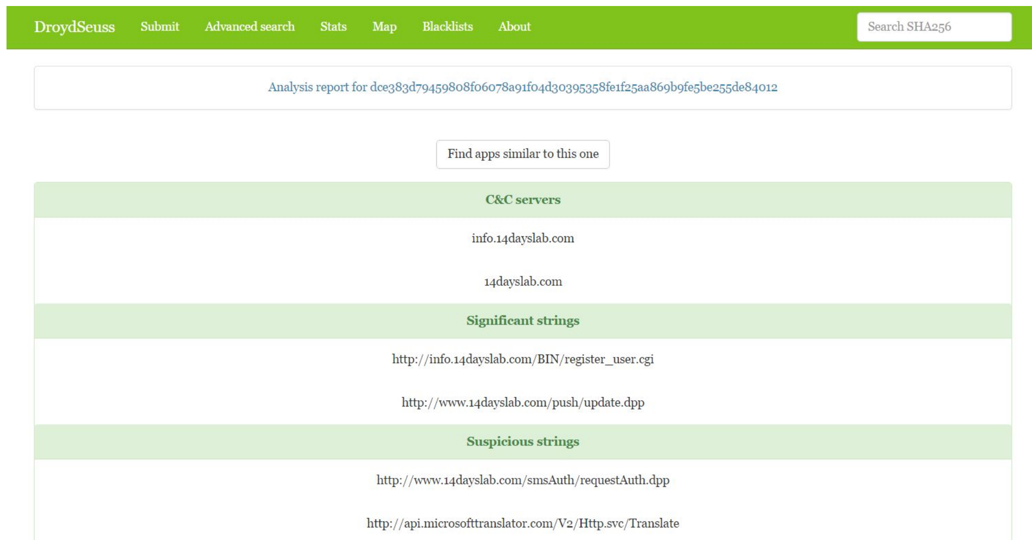


Figure 4.4: Analysis report

Figure 4.5: The result of a failed similarity search

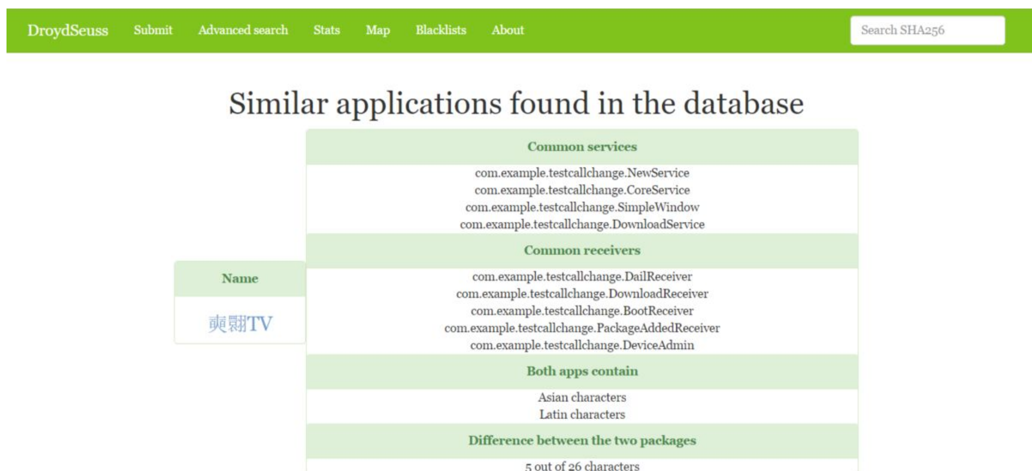


Figure 4.6: Similarity report showing correlated applications

Chapter 5

Experimental evaluation

Before releasing our work in the fully operative online web application, we analysed the scalability with a restricted number of applications stored, in order to predict the final responsiveness, and tested some use cases to prove the effectiveness. While inserting applications in the database, we downloaded them from the online malware repository KOODOUS[1], selecting on purpose the widest range of features. We chose legitimate apps from the same developer (e.g. Facebook, `com.facebook.katana`[2], Facebook Lite, `com.facebook.lite`[3], and Messenger, `com.facebook.orca`[4], are all produced by developer *Facebook Mobile*[5]); applications with the same icon; applications known to contain malicious functionalities; others with no requirement fulfilled. The results we present are collected on a database whose dimension doubled with every measurement, containing respectively 10, 20, 40 and 80 samples. We have an insight into the time metrics required by the features we developed in Section 5.1. We present the scenarios we tested and the conclusions we deduced in Section 5.2. We focus on which are or can potentially be limitations of our work in Section 5.3.

5.1 Performance

We are now focusing our attention on the empirical measurements of time consumption due to our work and we make predictions about them on the final database. In Section 5.1.1, we analyse the time required to fetch samples

[1]https://koodous.com/
[2]https://play.google.com/store/apps/details?id=com.facebook.katana
[3]https://play.google.com/store/apps/details?id=com.facebook.lite
[4]https://play.google.com/store/apps/details?id=com.facebook.orca
[5]https://play.google.com/store/apps/developer?id=Facebook

from the database. In Section 5.1.2, we investigate how much time is required to find applications correlated to a given one.

5.1.1 Retrieval from the database

We tested the time taken to fetch samples from the database by taking into account the most time consuming task: the retrieval of all the database in one single request. We first discuss the average time required to fulfil the request, then we analyse the variance as the number of samples increase and finally we present our prediction about the behaviour of our work on the real-life database.

Time measurements

As we can see in Table 5.1, DroydSeuss computations and scanning of the JSON document occupy a meagre amount of the total time taken to fulfil the search. In the case of 10 samples stored, it's 11.74%, then gradually decreases to 7.08% when the samples are 20, to 4.86% when the database size is 40 and finally to 3.77% in the last case, that is 80 apps stored. Each time a request is sent to Elasticsearch a fixed amount of time is employed to open the connection and receive the reply: this results in an ever reducing period of time spent to fetch each stored application. With 10 applications present, 5.5 milliseconds are used to retrieve each of them; then, doubling the amount to 20, only 4.8 milliseconds are needed. In the case of 40 apps, 4.6 milliseconds are required and in the case of 80 samples 3.3 milliseconds are necessary. In conclusion, even though the time needed to retrieve the whole database grows as the dimensions increase, it does so more slowly than a linear trend and is mainly due to the actual time Elasticsearch spends collecting the samples.

Standard deviation measurements

Increasing the number of stored applications may lead to a higher amount of variation or dispersion of the set of data values. This concept is quantified by the standard deviation, which is defined as in Formula 5.1.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \quad \text{where} \quad \mu = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{5.1}$$

We checked and verified that the standard deviation actually increases while fetching the entire database, but it is mainly caused by Elastic-

SEARCH non-uniform time taken to fetch. In Table 5.2, we can see that the standard deviation due to ELASTICSEARCH goes from 1.5X, while having 10 apps stored, up to 2.5X, in case of 80 applications; at the same time, the time occupied by DROYDSEUSS computation only increases by circa 1.6X each time, meaning that it augments more slowly than the increase in size of the database. The implications of the intensification of the standard deviation can be seen in Figure 5.1, where the vertical black bars represent the range in which all the measurements landed.

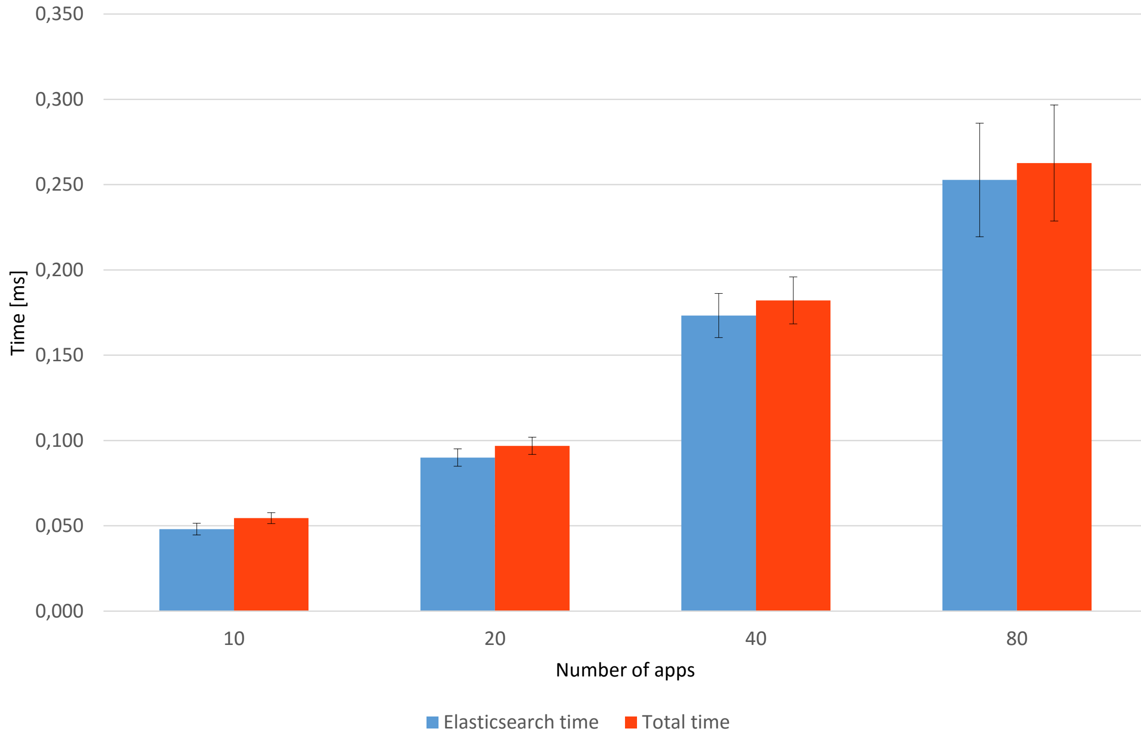


Figure 5.1: Time taken to retrieve the entire database

Performance predictions over the real database

Using a power trend function, defined as in Forumla 5.2, we predicted that in the real database, which consists of 1787 analysed samples, the time taken by DROYDSEUSS algorithms to retrieve the whole database will be 2.07% of the total time of 3.123 seconds, since the time taken by each application will be of 1.75 milliseconds.

$$f : x \mapsto cx^r \; c, r \in \mathbb{R}; 0 \leq c \leq 1 \tag{5.2}$$

	Time required [s]			
Number of samples	**Elasticsearch**	**DroydSeuss**	**Total**	**Time per app [ms]**
10	0.048	0.006	0.055	5.50
20	0.090	0.007	0.097	4.85
40	0.173	0.009	0.182	4.55
80	0.253	0.010	0.263	3.29

Table 5.1: Average time to search over the entire database

	Standard deviation		
Number of samples	**Elasticsearch**	**DroydSeuss**	**Total**
10	0.68%	0.14%	0.64%
20	1.02%	0.08%	1.00%
40	2.59%	0.22%	2.75%
80	6.66%	0.35%	6.80%

Table 5.2: Standard deviation on a search over the entire database

5.1.2 Find correlated applications

We take a look at the scalability of the database we set up for our tests when finding correlated applications. Since we chose applications with widest range of features, we expected to detect differences between the samples with related applications and the ones without. We first discuss the average time required to find the cluster of similarity, then we analyse the variance as the number of samples increases and finally we present our prediction about the behaviour of our work on the real-life database.

Time measurements

The nature of this feature implies that the more stored samples, the more clusters of similarity can be found. We noticed that even for applications that could be part of more clusters composed by more than two other apps the amount of time to fulfil the request did not significantly change. As we can see in Table 5.3, when we had 10 samples stored the average total amount of time required to find correlated applications was of 0.066 seconds, of which only 6.06% was due to DroydSeuss computations. Adding ten more samples, the average total time became 0.115 seconds, of which only 5.22% taken by DroydSeuss. Having 40 stored applications we got to 0.205 seconds taken in total, of which 6.83% used by our algorithms, and finally with 80 samples we reached 0.299 seconds employed to find clusters, of which

6.69% due to DroydSeuss computations. The first significant clusters of correlation began to show up when we got to 40 applications stored and increased as expected when we doubled the database size once again, but the average total time to complete the request only augmented from 1.5X to 1.75X, of which the amount taken by our algorithms was steadily between 5% and 7%. These achievements are reflected in a constantly reducing time taken to retrieve each app, which went from 6.6 milliseconds down to 3.7 milliseconds in the last case studied.

Standard deviation measurements

We calculated standard deviation (Formula 5.1) to understand how uniform we can consider the finding of correlated apps to be. As shown in Table 5.4, the variance takes a significant leap only when the database size gets to 80 samples, due to the fact that some applications had no correlations while others were already part of several clusters of considerable dimensions. However, it must be noted that the data dispersion is mainly due to the time variation that occurs in Elasticsearch to provide a response, since the computations made by our work have a variation of only 0.03%. Moreover, the standard variation is still less than the one measured for the case studied in Section 5.1.2.

Performance predictions over the real database

Using a conveniently tuned power trend function, defined as in Formula 5.2, we predicted that in the real database, which consists of 1787 analysed samples, the time taken by DroydSeuss algorithm to find the clusters of similarity of a given application will be 3.121 seconds. The variance is predicted to significantly increase up to 11.56%, due to computations needed to retrieve and scan the common components when correlated applications are found, but the time to fetch each stored sample is reduced to 1.7 milliseconds. Notably, the time required by DroydSeuss to read the JSON containing the needed information is expected to have a variance of 0.05%, meaning there is a very little difference between an application with correlations and one that is independent, as far as our work is concerned.

5.1.3 Overall performance conclusions

To sum up the empirical measurements we made, we discovered a correlation between the dimension of the online database, based of the number of samples

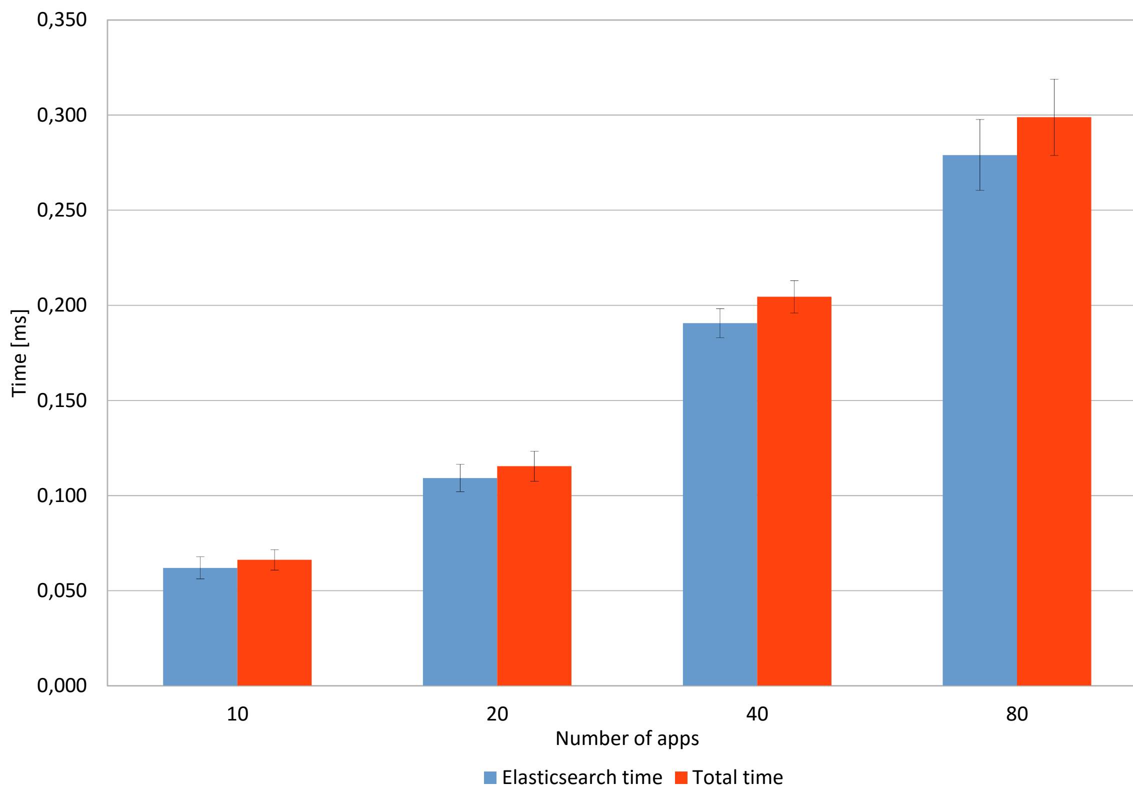


Figure 5.2: Time taken to find a cluster of similarity

contained, and the time required to fulfil the query on it. We can analyse this process splitting it in several tasks.

Connection to Elasticsearch This task has to be executed twice for each request, once to send it and once to retrieve the reply. It highly depends on the clock frequency of the processor used and the latency of the internet connection.

Query execution on Elasticsearch This task has a correlation with the database size (the bigger, the more time required) and can be improved only with an enhancement of the performance of the online service. As we can see in Figure 5.3, the dependence between database size and time required is not linear, as the time required to fetch each sample decreases as the overall size increases

DroydSeuss computations After the JSON reply from ELASTICSEARCH is retrieved, we need to parse it and apply the needed algorithms. This task is the only one we have the complete control of; however, as we

	Time required [s]			
Number of samples	**Elasticsearch**	**DroydSeuss**	**Total**	**Time per app [ms]**
10	0.062	0.004	0.066	6.60
20	0.109	0.006	0.115	5.75
40	0.191	0.014	0.205	5.13
80	0.279	0.020	0.299	3.74

Table 5.3: Average time to find correlated applications

	Standard deviation		
Number of samples	**Elasticsearch**	**DroydSeuss**	**Total**
10	1.16%	0.00%	1.07%
20	1.44%	0.00%	1.58%
40	1.53%	0.02%	1.70%
80	3.72%	0.03%	4.02%

Table 5.4: Standard deviation to find correlated applications

showed in Table 5.1 and in Table 5.3, the time consumption of this part of the process is minimal with respect to the overall amount required.

Since two out of three tasks require more time as the database size increases, but they do so slower than linearly, we expect the overall process to follow the same trend, up to the threshold in which the time to retrieve each sample cannot be reduced any more.

5.2 Use cases

In this section, we are presenting the use cases and scenarios we tested to prove the detection offered by our work is effective. Since we are presenting a new functionality to an existing analysis tool, it is not possible for us to provide a numerical evaluation of the improvement we got. Moreover, the features we offer rely on known characteristics of the malware and observed behaviours; as a consequence, we could pursue a qualitative inspection of our results of detection and not a quantitative one.

In Section 5.2.1, we focus on how we detected a campaign of 15 malware targeting the Asian market by detecting the similarity in their package names. In Section 5.2.2, we explain how we found similarities between apps apparently unrelated. In Section 5.2.3, we present the detection of a cluster of benign applications developed by the same company.

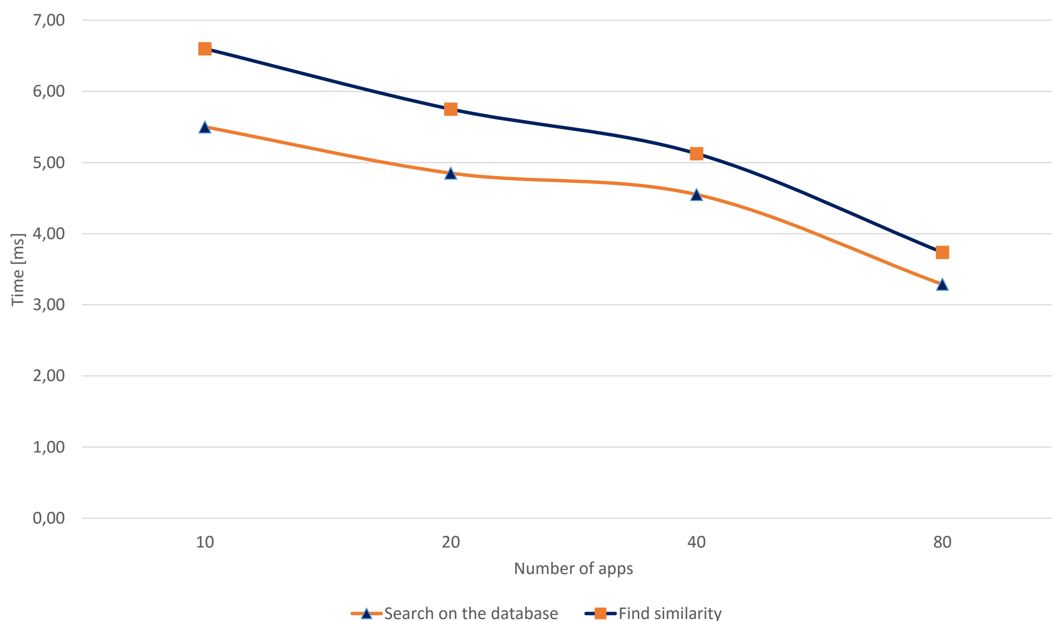


Figure 5.3: Time required per application stored when fetching the database

5.2.1 Detection of a malicious campaign by package similarity

We selected 15 applications found on Koodous which shared the same icon and part of the package name, but were apparenlty belonging to different development companies; however, each developer name was a random four character long string with no meaning. Without further information, we wanted to discover whether the apps were related, and thus part of the same malicious campaign, or not.

After the analysis, we got to know that the Levenshtein distance among the package names was on average equal to 8. The package structure was the one we presented in Section 4.3.2, in which there is a constant prefix, `com.shenqi.video.`, followed by two random four character sequences as "application name". Consequently, the threshold defined in Formula 4.2 was always met and thus the similarity check was triggered regardless the presence of common endpoints. In the end, the package pattern followed the structure `com.shenqi.video.*.*`, as it is detailed in Table 5.6.

In the data extracted from the static analysis, we discovered that all the

15 applications were actually the same, re-branded with different names and packages. They shared 13 activities, 9 receivers, 10 services, 33 filters, 33 permissions and, most notably, 3 malicious endpoints. More information about the data retrieved in the analysis can be found in Table 5.5. We concluded they are part of the same malicious campaign and DroydSeuss successfully grouped them all.

5.2.2 Detection of a malicious campaign by common malicious endpoints

As we detail in the preamble of Section 5, some of the applications we selected to populate our database were not chosen according to any prefixed requirement. We found out that two of them are probably generated by the same crimeware kit and belong to a single threat agent.

The applications under analysis seem to be both multimedia players, developed by two different companies; the details about their general information are listed in Table 5.7. After having stored their analysis, both of them contained a number of suspicious strings, thus we searched for similarities. We discovered they share one malicious and several suspicious endpoints, so the examination in order to find more correlated elements was triggered. The similarity report, as shown in Table 5.8, revealed they target the same markets and share 43 permissions, 5 suspicious endpoints, 1 malicious endpoint and 2 activities. It is however important to notice that the sample with package `tpkuvvaay.igtho.zk` contains more suspicious and malicious endpoints than the other application (most notably `api.taomike.com` and `cdn.1000su.com`). It suggests us that the two applications probably have been generated using the same crimeware kit but have been personalized in a different manner, therefore they might be under control of distinct threat agents.

This kind of information can be useful for future classifications: we are led to believe that if a sample shares a set of these characteristics, especially the activities `cn.rd.fastfun2.sdk` and `com.zhangzhifu.sdk.util.SimState` and the malicious endpoint `ad.jinchi168.com`, it belongs to a group of malware that exploit the same technique and consequently apply the selfsame countermeasure.

Activities	com.shenqi.video.Welcome com.haaa.util.ShowTipsActivity com.haaa.third.ThirdDialogActivity com.haaa.util.MyFormActivity com.shenqi.video.WebViewPlay com.haaa.util.Dialog1_3Activity com.shenqi.video.MainActivity com.haaa.util.Dialog2Activity com.shenqi.video.GuidViewActivity com.haaa.util.ShowProgressActivity com.android.os.AdActivity com.shenqi.video.DownLoadActivity com.cmcc.cmvsdk.main.NewOrderActivity
Malicious endpoints	zhxone.com lovev.com cmvideo.cn
Receivers	com.shenqi.video.paed.cwqz.MyBroadcastReceiver com.haaa.ivrCall.PhoneStateChangeReceive com.haaa.managePhone.PhoneStarBroadcast com.shenqi.video.PackageReciever com.shenqi.video.PhoneStarBroadcast com.haaa.sms.SmsReceiver com.haaa.managePhone.PhoneShutDownReceive com.android.os.BootReceiver com.haaa.apn.ConnectionChangeReceiver
Services	com.shenqi.video.FxService com.haaa.aidl.MyService com.haaa.task.TaskService com.haaa.managePhone.PhoneShutDownService com.haaa.managePhone.PhoneStarService com.haaa.managePhone.DealStartService com.haaa.apn.ConnectionChangeService com.shenqi.video.util.DownAPK com.android.os.Vpn com.haaa.mms.MmsService

Table 5.5: Extract of the data in common between the 15 samples

Application name	Developer name	Package name
忧色影院	yqab	com.shenqi.video.kjji.dioz
私密快播	fkyq	com.shenqi.video.knwg.wvvs
美妞影院	pbpl	com.shenqi.video.gjxf.pxmh
私密快播	vetp	com.shenqi.video.aval.grzq
美妞影院	dxje	com.shenqi.video.pdmb.rkwv
私密快播	otmf	com.shenqi.video.rkqq.edno
私密快播	krco	com.shenqi.video.visu.luyf
私密快播	dfno	com.shenqi.video.tocy.duzl
私密快播	acky	com.shenqi.video.qbju.puft
私密快播	pwyp	com.shenqi.video.veqf.wuxm
私密快播	jbew	com.shenqi.video.ymmb.wqlw
忧色影院	ozbg	com.shenqi.video.zbfi.ldbd
美妞影院	bzbv	com.shenqi.video.clgr.xxrh
私密快播	nrza	com.shenqi.video.weou.bqvy
婷婷影院	baal	com.shenqi.video.hwxc.uopz

Table 5.6: Samples from malicious campaign

Application name	Developer name	Package name
KMPlayer	aeYnKR	ovg.xtdq.hykgbttcfy
VivoPlayer	cWhKDy	tpkuvvaay.igtho.zk

Table 5.7: General information about the two samples

Malicious endpoints	ad.jinchi168.com
Suspicious endpoints	112.74.194.208 c22.cmvideo.cn mmsc.monternet.com mmsc.myuni.com.cn mmsc.vnet.mobi
Activities	cn.rd.fastfun2.sdk com.zhangzhifu.sdk.util.SimState

Table 5.8: Extract of the content shared by the two samples

Application name	Package name
Facebook	com.facebook.katana
Groups	com.facebook.groups
Lite	com.facebook.lite
Messenger	com.facebook.orca
Moments	com.facebook.moments
Pages Manager	com.facebook.pages.app
Work Chat	com.facebook.workchat

Table 5.9: Samples developed by *Facebook Mobile* we considered

5.2.3 Grouping applications of the same benign developer

In this scenario, we used the applications developed by *Facebook Mobile* [6] listed in Table 5.9, a very popular non malicious company, to show how our work can find correlations and group together submitted samples, even if harmless. This allows to categorize a developer as benign and insert it in a trusted list.

Analysing these applications we did not find malicious C&C servers in common, but since the developer followed the package naming conventions described in Section 4.3.2, DroydSeuss recognized the similarity and started the search for clusters anyway. All the samples share the common prefix `com.facebook.*`, which is 13 characters long, and are followed by their application identifier, which ranges between 4 and 9 characters; the average Levenshtein distance is 6.26, from a minimum of 4 to a maximum of 8. Consequently, the condition defined in Formula 4.1 was met in all cases.

The results are coherent with the premise we set. We expected the applications to share several characteristics, since they come from the same company, but not to contain malicious endpoints. We discovered that all the samples have GCM connectivity, share a set of permissions (some of which are considered potentially harmful by Google, e.g., the possibility to change the device connectivity state, the write access to the main memory, etc...) and the connection to a URL hosted by a domain of the company itself. Details about the most significant shared information we extracted are listed in Table 5.10 .

[6] https://www.facebook.com/

Permissions	android.permission.ACCESS_COARSE_LOCATION
	android.permission.ACCESS_FINE_LOCATION
	android.permission.ACCESS_NETWORK_STATE
	android.permission.ACCESS_WIFI_STATE
	android.permission.BATTERY_STATS
	android.permission.CAMERA
	android.permission.INTERNET
	android.permission.READ_PHONE_STATE
	android.permission.VIBRATE
	android.permission.WAKE_LOCK
	android.permission.WRITE_EXTERNAL_STORAGE
Intents	com.android.launcher.permission.INSTALL_SHORTCUT
	android.intent.action.BOOT_COMPLETED
	android.intent.action.SEND
	android.net.conn.CONNECTIVITY_CHANGE
	android.net.conn.com.facebook.GET_PHONE_ID
GCM	com.google.android.c2dm.intent.RECEIVE
	com.google.android.c2dm.intent.REGISTRATION
	com.google.android.c2dm.permission.RECEIVE
Domains	https://b-api.facebook.com/
	http://www.android.com/

Table 5.10: The most significant shared information extracted from *Facebook Mobile* apps

5.3 Limitations

In Section 5.3.1, we explain which are the issues inherited from using dynamic analysis in a sandbox. In Section 5.3.2, we focus on the necessity of a human check over the final output result. Finally, in Section 5.3.3, we hypothesize the need of a revision of the predictions over the database time performances.

5.3.1 Sandbox limitations

Having to rely on dynamic analysis in a sandbox, DroydSeuss is vulnerable to the restrictions of this approach. The modern generation of malware is capable of looking like acceptable code and tricking the sandbox to gain system access. After getting by the sandbox, the malware first lies dormant so that it does not trigger suspicions, then changes its characteristics revealing its true nature. [24] Sandboxes must be able to monitor activity to protect against this latest generation of threats. Threat agents can fingerprint the environment and design code that on purpose evades the detection provided by DroydSeuss. If that were the case, a radical change in our tool structure would be essential or, alternatively, the analysis would need to be done on real devices.

Another issue inherited from sandboxes design is code coverage. In order to detect a specific behaviour, the code path that triggers it must be stimulated by the runtime simulation and some sequence patterns may be not reached. We need to rely on TraceDroid ability to initiate the communication with a C&C endpoint to detect it.

5.3.2 Human supervision

Despite being a useful and reliable tool to avoid manual inspection over the malware code, human verification is necessary in order to surely classify a cluster of applications as malicious. Because of the algorithm implementation we explained in Section 4.3, legitimate applications created by the same developer are classified as a group. If they don't share suspicious or malicious endpoints, they can be easily discarded from the collection of malware; however, if they share even just one suspicious endpoint, which could be a false positive as we explained in Section 4.1.2, human review is required in order to be sure whether to categorize their developer among the trusted ones or the malicious ones.

5.3.3 Real database performance

In Section 5.1.1 and Section 5.1.2, we presented the analysis of the performances we achieved when testing our work on a database created according to our needs. For each measurement, we doubled the number of stored applications, up to having 80 samples in the last one. We made mathematical predictions using the data we collected and the two models we obtain seem to be coherent with each other; however, the real database on which our work will be employed is slightly more than one order of magnitude bigger than the one used for the testing. It is possible that the standard deviation, the most difficult quantity to estimate due to the amount of variables that influence it, differs from the foreseen one, especially when searching for similarities; in such a case, the model should be revised to better predict the future of the performances.

Chapter 6

Conclusions

In this chapter, we present our conclusions regarding Android malware campaigns recognition. We first propose a set of future research efforts in Section 6.1, followed by a conclusive overview of our contributions in Section 6.2.

6.1 Future work

In Section 6.1.1, we detail which data we excluded from our algorithms and thus may be used in future researches. In Section 6.1.2, we outline how the data we collect can be useful to an ISP provider to prevent malicious attacks to its customers.

6.1.1 Usage of `strings` and `classes`

During our extension of the amount of data extracted from the source code, described in Section 4.1.1, we decided to get the `strings` and the `classes` of the applications. However, when working on the algorithms to detect similarities described in Section 4.1.2, we couldn't find any additional value in including those pieces of information in the analysis. Here are the motivations that led to their exclusion.

- In Section 4.3.1, we detailed how we were searching for the presence of shared C&C servers between the applications. Including the `strings` in this comparison would have led to terribly misleading results, since some words of common use are likely to be retrievable from every sample. A similar reasoning was made about the `class` structure: it is handy to have files called *Homepage*, *Settings* or *Payment* to easily spot the functionality of a certain `class` when developing, thus finding a false connection between unrelated apps.

- The metrics explained in Section 4.3.2 is related to the package names, thus disconnected to the concepts of `string` and `class`.
- In the final report of similarity, presented in Section 4.4.2, the amount of data presented is rich in information, especially for applications made by the same developer. We preferred to focus the users attention on the significance of that material instead of overwhelming their knowledge with data with correlation.

These reasons are intrinsic of the metrics and approach we decided to follow while designing our work. Other researches have successfully used the `class` tree structure as the focal point of their work [12]. Furthermore, we based our clustering algorithms on empirical observations made on the reports available from the applications analysed by the original version of DroydSeuss; it is possible that having more data added to the samples stored could lead to a more refined and efficient cluster recognition or usage of the collected information. A future path of research could integrate the current state of the work with the results of both proposed new approaches, in order to achieve a higher clustering precision.

6.1.2 Automatic detection rules

The data we collect about clusters' characteristics can be useful to remotely block malicious connections. Two different approaches are possible.

- Each time a new malicious endpoint is detected, it is added to a blacklist of connections to be avoided. The ISP provider can continuously check the list and when a device that uses its network connects to a blacklisted domain, deny all the data transfer. This is a very cautious strategy that is time intensive if the list is long.
- A more sophisticated solution is to store the information about the malicious endpoints used by each clusters, every time one is detected; if a device connected to the network of the ISP provider employing this strategy wants to connect to a malicious domain, the request is blocked and pre-emptively all the future connections to the other endpoints part of the same cluster are denied. Moreover, if a significant number of devices are under attack of the same malware campaign, the provider can decide to disallow to all users the connection to the above mentioned endpoints.

The study about the utility and feasibility of such strategy are left for future researches.

6.2 Conclusions

The devices sales, the number of applications on the Google Play Store and the number of applications downloaded are all expected to increase until 2020, thus it's reasonable to foresee the malware growth to follow the same trend. The never-ending increase of Android's market share has brought a conspicuous number of cyber criminals to develop malware against it. The mitigations deployed by Google have proved to be as not effective enough.

DroydSeuss is a malware tracker that helps security analysts by automatically analysing Android applications, avoiding the necessity of a manual inspection. Its primary focus is on banking trojans, a class of malware with the goal of intercepting the data when users connect to their bank account, stealing private information with a man-in-the-middle attack. The threat agents must communicate with a C&C endpoint that they control, usually a web domain or a telephone number, from which the malware receives instructions and to which it sends the stolen data. Tracking the connections to the network, we can detect malicious behaviours performed by the malware.

We introduced a new approach and developed DroydSeuss features in order to mitigate the malware threat.

- We developed an advanced search engine, to allow the users to browse the database of already analysed samples, with parametrized queries. We employed Elasticsearch and its Python libraries for both full-text indexing and remote communication. These aspects of our work are detailed in Section 4.2.

- We developed an algorithm to find correlations between analysed applications. To provide reliable and useful information, we first needed to extend the amount of data extracted from the source code and the analysis over it, as explained in Section 4.1. Having the possibility to group samples into clusters of similarity allows to detect which apps were created using the same crimeware kit or were developed by the same company. With these new pieces of information collected, it is possible to make predictions about new malicious applications found. Further information about this innovation is available in Section 4.3

Our challenge was to provide these feature as part of a responsive web application. We paid a lot of attention to minimize the time consumption, in order to fulfil the users' request in almost real-time. To have an insight on the time required by each task, we detailed the achievements we accomplished in Chapter 5. We reached a stage where only less than 4% of the total time required is due to computations made by our work and the estimations about

the scalability in the real-life database predict the overall impact will still be minimum.

Since the practical goal of our work was to add features to an existing tool, we could not provide a quantitative analysis of our results, but we rather focused on having a qualitative one. In Section 5.2, we presented the results of three different use cases we modelled: the recognition of a cluster of malware by package similarity; the recognition of a cluster of malware by common endpoints shared; the classification of a group of benign applications of the same legitimate developer. We demonstrated the feasibility and reliability of the intelligence algorithms we developed and in Section 6.1 we proposed future paths of research based on different assumptions and different data usages.

Bibliography

[1] Shay Banon and Steven Schuurman. Elasticsearch - Search & Analyze Data in Real Time, Feb. 2010. 2, 14

[2] Alberto Coletta, Victor Van Der Veen, and Federico Maggi. DroydSeuss: A Mobile Banking Trojan Tracker - Short Paper. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, Feb. 2016. 1

[3] Anthony Desnos. Android : Static Analysis Using Similarity Distance, 2012. 30

[4] Anthony Desnos and Geoffroy Gueguen. Android: From Reversing to Decompilation. `https://github.com/androguard/androguard`, Dec. 2011. 2, 12, 19

[5] Internet Engineering Task Force. DARPA Internet Program Protocol Specification. `https://tools.ietf.org/html/rfc791`, Sep. 1981. 21

[6] Richard W Hamming. Error detecting and error correcting codes, Apr. 1950. 30

[7] Hispasec. VirusTotal - Free Online Virus, Malware and URL Scanner, Jun. 2004. 2, 14

[8] Natalia Stakhanova Hugo Gonzalez and Ali A. Ghorbani. DroidKin: Lightweight Detection of AndroidApps Similarity, Jan. 2015. 30

[9] IDC. IDC Worldwide Quarterly Mobile Phone Tracker, Mar. 2016. 1, 5

[10] Xuxian Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2. `http://www.cs.ncsu.edu/faculty/jiang/appverify`, Dec. 2012. 1

[11] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 163: 845–848, 1965. 2, 25, 30

[12] Saung Li. Juxtapp and dstruct: Detection of similarity among android applications. Master's thesis, EECS Department, University of California, Berkeley, May 2012. 30, 52

[13] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer, Jun. 2012. 1

[14] Nicholas J. Percoco and Sean Schulte. Adventures in BouncerLand, Failures of Automated Malware Detection within Mobile Application Markets, 2012. 5

[15] Thu Pham. Answer to OTP Bypass: Out-of-Band Two-Factor Authentication, Jul. 2014. 6

[16] Matías Porolli and Pablo Ramos. CPL Malware in Brazil: Somewhere Between Banking Trojans and Malicious Emails, May 2014. 5

[17] RiskIQ. RiskIQ Reports Malicious Mobile Apps in Google Play Have Spiked Nearly 400 Percent, Feb. 2014. 1, 5

[18] Lukáš Štefanko Robert Lipovský and Gabriel Braniša. The Rise of Android Ransomware, Feb. 2016. 6

[19] SCMagazine. Google using custom malware scanner for Android apps, Feb. 2012. 1

[20] Statista. Number of mobile app downloads worldwide from 2009 to 2017 (in millions), Jul. 2014. 1, 5

[21] Statista. Number of apps available in leading app stores as of July 2015, Jul. 2015. 1, 5

[22] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 4th quarter 2015, Jan. 2016. 1, 5

[23] Victor Van Der Veen. Dynamic Analysis of Android Malware. Master's thesis, VU University Amsterdam, Aug. 2013. 7

[24] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection, 2014. 49

Appendices

Appendix A

Elasticsearch mapping

Listing A.1 shows the structure we used to map the data extracted from the analysed applications on the ELASTICSEARCH online database.

Listing A.1: Elasticsearch mapping

```
droid-1
{
  "mappings": {
     "app": {
        "properties": {
           "GCM_ID": {
              "type": "string"
           },
           "activities": {
              "type": "string",
              "fields": {
                 "raw": {
                    "type": "string",
                    "index": "not_analyzed"
                 }
              }
           },
           "classes": {
              "type": "string",
              "fields": {
                 "raw": {
                    "type": "string",
                    "index": "not_analyzed"
                 }
              }
           },
           "endpoints": {
              "type": "string",
              "fields": {
```

```
      "raw": {
         "type": "string",
         "index": "not_analyzed"
      }
   }
},
"hasArabic": {
   "type": "boolean"
},
"hasCJK": {
   "type": "boolean"
},
"hasCyrillic": {
   "type": "boolean"
},
"hasGreek": {
   "type": "boolean"
},
"hasLatin": {
   "type": "boolean"
},
"ips": {
   "type": "string"
},
"malicious_domains": {
   "type": "string",
   "fields": {
      "raw": {
         "type": "string",
         "index": "not_analyzed"
      }
   }
},
"minSDK": {
   "type": "integer"
},
"name": {
   "type": "string"
},
"num_end": {
   "type": "string"
},
"num_suspicious": {
   "type": "string"
},
"numbers": {
   "type": "string"
},
"package": {
```

```
      "type": "string",
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "permissions": {
      "type": "string",
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "receivers": {
      "type": "string",
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "safe_domains": {
      "type": "string",
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "services": {
      "type": "string",
      "fields": {
        "raw": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    },
    "sha256": {
      "type": "string"
    },
    "significant": {
     "type": "string",
```

```
              "fields": {
                "raw": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            },
            "strings": {
              "type": "string",
              "fields": {
                "raw": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            },
            "suspicious": {
              "type": "string",
              "fields": {
                "raw": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            },
            "suspicious_domains": {
              "type": "string",
              "fields": {
                "raw": {
                  "type": "string",
                  "index": "not_analyzed"
                }
              }
            },
            "use_GCM": {
              "type": "string"
            }
          }
        }
      }
    }
```

Appendix B

Regexen to filter endpoints

Listing B.1 shows the regular expression we defined in order to extract the domain from a URL endpoint retrieved in the static analysis.

Listing B.1: Regex definition of a domain

```
(http(s)?:\/\/)?([-0-9A-Za-z\x80-\xFF]+\.)+(com|net|org|int|edu|
gov|mil|arpa|aero|biz|coop|info|museum|pro|cat|jobs|travel|mobi|
ac|ad|ae|af|ag|ai|al|am|an|ao|aq|ar|as|at|au|aw|ax|az|ba|bb|bd|be|
bf|bg|bh|bi|bj|bm|bn|bo|br|bs|bt|bv|bw|by|bz|ca|cc|cd|cf|cg|ch|ci|
ck|cl|cm|cn|co|cr|cs|cu|cv|cw|cx|cy|cz|dd|de|dj|dk|dm|do|dz|ec|ee|
eg|eh|er|es|et|eu|fi|fj|fk|fm|fo|fr|ga|gb|gd|ge|gf|gg|gh|gi|gl|gm|
gn|gp|gq|gr|gs|gt|gu|gw|gy|hk|hm|hn|hr|ht|hu|id|ie|il|im|in|io|iq|
ir|is|it|je|jm|jo|jp|ke|kg|kh|ki|km|kn|kp|kr|kw|ky|kz|la|lb|lc|li|
lk|lr|ls|lt|lu|lv|ly|ma|mc|md|me|mg|mh|mk|ml|mm|mn|mo|mp|mq|mr|ms|
mt|mu|mv|mx|my|mz|na|nc|ne|nf|ng|ni|nl|no|np|nr|nu|nz|om|pa|pe|pf|
pg|ph|pk|pl|pm|pn|pr|ps|pt|pw|py|qa|re|ro|rs|ru|rw|sa|sb|sc|sd|se|
sg|sh|si|sj|sk|sl|sm|sn|so|sr|ss|st|su|sv|sx|sy|sz|tc|td|tf|tg|th|
tj|tk|tl|tm|tn|to|tp|tr|tt|tv|tw|tz|ua|ug|uk|us|uy|uz|va|vc|ve|vg|
vi|vn|vu|wf|ws|ye|yt|yu|za|zm|zw)
```

Listing B.2 shows the regular expression we defined to recognize if an endpoint is an IP.

Listing B.2: Regex definition of an IP

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]
?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]
|2[0-4][0-9]|[01]?[0-9][0-9]?)(:[0-9]+)?$
```