

Dibris

Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi (DIBRIS)

Appregator: a Large-Scale Platform for Mobile Security Analysis

by

Federico Lucini

Master Thesis

DIBRIS, Università di Genova

Via Opera Pia, 13 16145 Genova, Italy

<http://www.dibris.unige.it/>

Università degli Studi di Genova



**Dipartimento di Informatica, Bioingegneria,
Robotica e Ingegneria dei Sistemi (DIBRIS)**

MSc Computer Science
Data Science and Engineering

**Appregator: a Large-Scale Platform for
Mobile Security Analysis**

by

Federico Lucini

Advisor: Alessio Merlo

Examiner: Giovanni Lagorio

Tutor: Luca Verderame

July, 2020

Table of Contents

Abstract	6
Chapter 1 Introduction	7
1.1 Contributions	7
1.2 Thesis Structure	8
Chapter 2 Background and Related Work	9
2.1 Android Architecture	9
2.2 Vulnerability Analysis	11
2.3 Tools and Datasets	13
Chapter 3 Requirements Analysis	14
3.1 Requested Features	14
3.1.1 Applications Collection and Update	15
3.1.2 Security Analysis	15
3.2 Requirements	17
3.2.1 Table View	17
3.2.2 Additional Studies	18
Chapter 4 Design	19
4.1 Architecture	19
4.1.1 Workflow	19

4.2	Crawler	20
4.3	Worker	21
4.4	Database	23
4.4.1	Typical Workload	23
4.4.2	Logical Schema	24
4.5	Server	24
Chapter 5	Implementation	26
5.1	Technologies	26
5.1.1	Approver	27
5.1.2	MongoDB	28
5.1.3	NodeJS and Docker	28
5.2	Technical Details	31
5.2.1	Crawler and Worker	31
5.2.2	Database and Server	33
Chapter 6	Experiments	35
6.1	Experimental Setup	35
6.2	Query Capabilities	35
6.2.1	Privacy	35
6.2.2	Security	39
6.3	Statistics	42
6.4	Additional Studies	47
Chapter 7	Conclusion	49
7.1	Discussion	49
7.2	Future Work	50
Bibliography		52

Acknowledgement

In apertura alla tesi, vorrei dedicare un sentito ringraziamento a tutte le persone che hanno agevolato il mio percorso: mi rivolgo a tutti i miei familiari e, in particolare, ai miei genitori ed a mio fratello che mi hanno sostenuto in questi anni. Anche il supporto dei miei amici è stato fondamentale e per questo ringrazio Francesco, Sebastian e Fabio.

Sono grato all'Università di Genova per l'offerta formativa e soprattutto al DIBRIS ed ai suoi docenti per l'organizzazione del corso di studio. Un doveroso ringraziamento va inoltre al professore Alessio Merlo e Luca Verderame per avermi dato l'opportunità di mettermi alla prova con l'argomento di questa tesi, e di conseguenza anche ai loro collaboratori Davide e Gabriel per l'aiuto che mi hanno prestato in questi mesi.

Abstract

Many tools perform automatic security analysis on Android apps to study a particular issue or find malware, but there are no tools that systematically retrieve aggregate statistics from app stores and compare different versions of the same app. Appregator is a large scale platform for automatic mobile security analysis, that aims to study the vulnerabilities which affect the most installed Android applications. It performs general-purpose security measures and allows researchers to extract and filter results from the analysis, to retrieve the list of apps with specific patterns or find the most common interesting scenarios for security or privacy. To test its performance and accuracy we collected information from almost 200.000 applications in the Play Store and performed security analysis on the top 2000.

Chapter 1

Introduction

Risk assessment is crucial for IT systems and it became a key issue also for mobile applications, which are increasingly used every day for private or working activities. To deal with app security problems, several tools for the analysis of vulnerabilities have been proposed over the years. Despite this, such efforts did not provide a solution allowing to build an overall view of the security status over a set of arbitrary mobile applications.

In particular, some research projects produced datasets that allowed to perform analysis on the security features of applications but they mainly contain malware. Indeed, automatic techniques are especially used to study malware and prevent its spread, but the same effort is not applied to find vulnerabilities exposed in applications that could be exploited both by malicious apps or external attackers.

Albeit there exist some proposals that allow to assess the number of applications affected by a specific vulnerability, the results and the tools are kept mostly private, so that only few information are available and the tools cannot be used to replicate the results in the wild.

1.1 Contributions

To overcome previous limitations, we propose the design and implementation of a large scale platform for mobile security analysis, with the purpose to help researchers discovering new vulnerabilities on Android applications.

The proposed platform is able to collect information about a big number of Android applications and perform security analysis on the most relevant ones, in a full automatic way and checking for updates. It allows researchers to extract and filter results which could be interesting for privacy or security reasons, and to obtain the list of applications that match a given set of conditions.

Our priority is to study vulnerabilities that affect many users, and therefore we refer to the most installed applications obtained only from the main stores: we realized and tested a prototype for the official Google Play Store market and the alternative Aptoide store.

In our experiments, we carried out a statistical security analysis, thereby showing how the platform can be used in actual security scenarios: to this aim, some examples retrieved from the Play Store, concerning the usage of insecure connections, privacy leaks and hard-coded keys or passwords will be discussed.

Moreover, we will show how the platform can be used to compare applications downloaded from different markets, detect modification between different versions of the same app, and perform statistical risk studies.

1.2 Thesis Structure

After an overview of the Android architecture and a survey on the main large scale studies on mobile security, we discuss automatic techniques and available tools for security analysis (Chapter 2) and rely on them to define which features must be taken into consideration, as well as the thesis goals (Chapter 3).

Starting from the set of requirements, we propose an architecture (Chapter 4) and discuss its implementation giving details about the adopted technologies (Chapter 5). In the next section (Chapter 6), we perform experiments on a set of actual applications, and provide some interesting scenarios and useful statistics. To conclude, we discuss the obtained results and point out some future work (Chapter 7).

Chapter 2

Background and Related Work

In this chapter, we provide an overview of Android architecture, an introduction to vulnerability analysis techniques, and we report available large scale studies on mobile security.

2.1 Android Architecture

Android Operating System is organized in four layers as shown in Figure 2.1, the colors identify different technologies: red components are related to the Linux kernel, green to native code (C/C++) and blue are associated with the Java language.

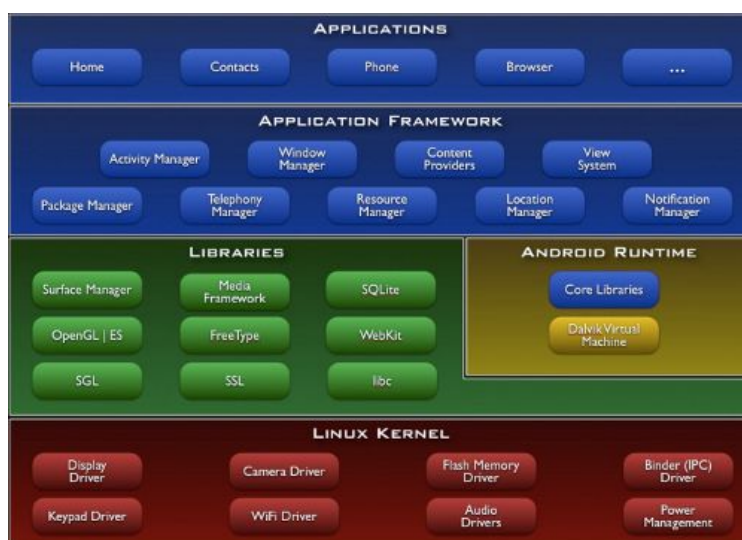


Figure 2.1: Android OS Architecture

We rely on *Android Operating System Architecture* article [Far18] to describe each layer.

- The first one provides system functionalities like process, memory, and device (camera, display) management. It is based on the Linux Kernel with the addition of an alarm driver to schedule operations, a binder to handle inter-process communication without using replication, but passing objects by reference, and power management suitable for smartphone usage. The OOM (out of memory) handler kills processes based on defined policies, the ashmem component supports low memory device discarding shared memory when needed, and a kernel debugger and logger are provided.
- The second layer includes some native libraries like a small size implementation of the libc, the SQLite relational database, the WebKit browser engine, and a Media Framework for recording and playing audio, video or pictures. The Android Runtime consists of core Java libraries and the Dalvik Virtual Machine, an alternative to the Java Virtual Machine designed for small size RAM and slow CPU systems.
- The Application Framework provides APIs as Java classes that can be requested by an application, for example using the telephone service to perform calls and send messages or the location service to access GPS and cell tower.
- The top layer includes pre-installed applications like a web browser, SMS client and contact manager or third party applications, usually downloaded from the Google Play Store. An application can contain activities that are screens with a user interface, background services to perform working operations, broadcast receivers to allow the system to deliver events to the app outside its regular usage and content providers which manage data shared in the persistent storage location like the file system or the SQLite database.

Once the kernel is started it setups cache, protected memory, scheduling and it loads hardware drivers, then the init root process mounts directories and provides user-defined policies or initialization instructions. The zygote process loads the core library classes, it forks another process to launch system services, the activity manager (responsible for the app lifecycle), the home screen launcher and it creates a new virtual machine instance for each application helping code sharing across the Dalvik VM.

Android implements mandatory access control in the form of permission system: an application can request the user to access system resources, but giving unnecessary permissions to an app is a security-related issue. The Linux kernel implements a monolithic architecture where no isolation between components (device drivers, shared memory, etc.) is provided, and therefore any exploit at this level is very dangerous because it allows modifying kernel memory: devices manufactures use custom drivers that could contain security bugs and the code is not shared with the Linux community. When a user roots the device, kernel integrity is compromised and android security measures might be disabled.

2.2 Vulnerability Analysis

Vulnerability analysis is performed by an automatic tool that scans a target, extracts its information and tries to identify security problems. The identification of these risk factors has some limitations due to false-positive results related to a wrong classified behavior or issues that are present but are not exploitable in the system. The job of a specialist who can evaluate these results is critical, but from a legal point of view he should be explicitly asked to test the security of a system, while many works simply make an automatic estimate without providing details to a potential attacker and therefore not being legally risky¹.

On the other hand, problems that could emerge from the combination of multiple vulnerabilities are usually not considered, as well as some security checks might not be taken into account by the tool. Both these aspects are problematic for a developer who needs to have most of the issues reported without manually analyze a large number of scenarios, however the main problem is that often security assessment is not even performed: only 29 percent of mobile applications are tested for vulnerabilities [BIT17].

There are two approaches to detect vulnerabilities in an application: static and dynamic analysis. The first one is more efficient, but it just examines the code without running the app, unlike the other. The execution of the code gives a more detailed idea about the application behavior, but it is difficult to emulate all the possible input actions and have a whole coverage of test cases because of the time required and the limitations of automatic methods to be applied on different apps. On the contrary, static analysis often considers test cases that are not actually performed, due to the presence of third-party libraries and services.

We have compared two studies [AEM⁺19, DD16] to understand what are the typical steps in the automatic analysis of Android applications, a description of each follows.

- **APK Reverse Engineering:** an Android Package (APK) is an archive containing the program's code, resources, certificate, and manifest². When a developer writes an app, its Java source code is compiled to JVM bytecode and then translated to .dex files, executed by the Dalvik VM until Android 4.4 or Android Runtime (ART) from Android 5.0³. In this step, the code inside *classes.dex* file in the archive is reversed and reconverted to JAR files. The most used software for this purpose are ApkAnalyzer, dex2jar or apktool that returns smali code, a human-readable representation of the dex format (the name smali is the Icelandic equivalent of assembler because Dalvik was named for an Icelandic fishing village⁴).

¹<https://www.eff.org/issues/coders/vulnerability-reporting-faq#faq3>

²https://en.wikipedia.org/wiki/Android_application_package

³[https://en.wikipedia.org/wiki/Dalvik_\(software\)](https://en.wikipedia.org/wiki/Dalvik_(software))

⁴<https://github.com/JesusFreke/smali/wiki>

- **Decoding Manifest File:** the manifest file is in binary format, it is converted to user-readable XML and extracted the main information like the application name, version, components, permissions, minimum and target SDK.
- **Static Analysis:** the code is analyzed to obtain the application life-cycle, emulated constructing a call graph between its components and Android APIs. The main tools are FlowDroid that generates a graph with methods as nodes and calls as arrows and the SuSi framework that classifies an Android API to be a source or a sink.
- **Dynamic Analysis:** the application is installed in a virtual machine and the user behavior is simulated using software like Droidbot, in the meantime network traffic is sniffed. Requested URLs could also be obtained filtering network API calls using Strace or tools like TraceDroid useful to log invoked Java methods with parameter resolution.

We rely on *Static Analysis of Android Apps: A Systematic Literature Review* [LBP⁺17] to describe different static analysis techniques, the main approaches are:

1. Control-flow, to consider all possible execution paths of a program.
2. Data-flow, to compute the set of possible values for every point of the program.
3. Points-to, to establish the data to which a variable can point.

A program starts with an entry point and by its inspection are retrieved the methods that it calls, but different concerns impact the algorithms used to construct the call graph. A flow-sensitive approach is aware of the order of program statements and produces less false positives of a flow-insensitive graph, which only lists APIs. A path-sensitive procedure produces a graph for each path, while an insensitive one considers all the methods in a single graph, the choice depends on the program dimensions. At last, the field-sensitive model considers the fields for each object, contrary to the field-insensitive approach that classifies class of objects. In addition to construction techniques, also information about the context of method calls is considered: in a context-sensitive analysis each method call is modeled independently, in particular, the object-sensitive approach distinguishes invocations of methods made on different objects.

The main challenge in Android analysis is the presence of several entry points, and therefore many graphs are considered with no assurance about their connections because of the life cycle of each component, UI actions or system events, Inter-Component Communication (ICC) and third-party libraries. Other limitations are related to the Java language, like the usage of reflections or the dynamic code loading, which are difficult to statically handle because classes could be generated at runtime and therefore the dynamic analysis is the only option.

2.3 Tools and Datasets

The tools mentioned in the previous section are often used as building blocks for other frameworks to analyze ICC abuse by malicious apps (in *SCanDroid*, *ComDroid*, *CHEX*, *Woodpecker*, *DroidChecker*, *DroidSafe*, *AppAudit*) or to monitor applications behavior (in *ProfileDroid*, *CopperDroid*, *Profiler*, *AppIntent*, *DroidMiner*, *AsDroid*). However, obfuscation strategy can often disturb assumptions on which these tools rely on: a malware could detect the analysis environment, perform malicious activities only at a certain point in time, adopt entry point pollution and anti-tainting techniques to evade dynamic tests or prevent static analysis by breaking use-def chains, using call graph degeneration, hiding types and bypassing signature matching mechanisms [HRM⁺16].

Another limitation is related to the absence of accurate information on tools configuration, which makes difficult to study when security flags are detected correctly [QWR18]. Indeed, comparing programs under different setup brings to irreproducible and inaccurate results.

Even Google developed a security mechanism known as Bouncer: an applications review process that verifies Google Play policies and provides security services automatically included as part of GMS (Google Mobile Services), like the app verification, network protection, and safe browsing, smart lock and lost or stolen devices defense [KJ18].

Datasets of Android Applications: a Literature Review [GM18] shows that there are five datasets that collect information from more than 1 million applications: Appannie⁵, AppBrain⁶, AndroZoo, AndroVault, and Andrubis. While the first two are commercial data aggregation tools that use the information to study clients' competitors and improve their market position, the others have a research purpose and provide collections of apps from which also metadata is extracted, thanks to static analysis tools.

Among the information collected in AndroZoo [LGH⁺17], there are those coming from the store, the manifest as the permissions, the *classes.dex* file as the presence of native and crypto code or reports of antivirus and vulnerabilities detection services. Meng et al. [MXS⁺17] claim that it is complex to extract information interesting for research from the AndroZoo dataset and therefore they build a graph of attributes to extract these data.

Androvault and Andrubis [PLN⁺14] also perform dynamic analysis, but in the first one [MXS⁺17], the authors were unable to collect the dynamic execution of many apps because of time costs. The last one instead is discontinued and data is gathered until 2015, and therefore obsolete and probably no more representative of Android markets. The main purpose of these works is malware recognition and for this reason, apps are obtained also from many alternative stores, virus collections or previously generated set.

⁵<https://www.appannie.com/>

⁶<https://www.appbrain.com/>

Chapter 3

Requirements Analysis

In this chapter, we analyze which are the features to implement and provide a list of the thesis objectives.

3.1 Requested Features

The goal of the thesis is the design and implementation of architecture able to:

1. automatically collect large set of mobile applications from app stores,
2. perform static and dynamic analysis of those applications,
3. store all the security-relevant information of the analysis to build a large security dataset that can be queried and used to get statistical results, and
4. automatically check the release of application updates to create a security history of mobile apps releases.

Unlike most of the related works presented in Section 2.3, the focus of this architecture is not the study of malware. Indeed, this work is focused on evaluating the security and privacy of mobile applications in general, being malware, or benign.

3.1.1 Applications Collection and Update

To the best of our knowledge, there is no large-scale study that implements an app collection system with the purpose to find vulnerabilities which affect most of the users.

The projects identified in Section 2.3 save security information, but the metrics are mainly applied to alternative stores or malware collections. Consequently, the architecture needs to rely on the evaluation of the most used apps available from the leading Android application stores: Google Play and Aptoide¹, which are respectively the official and the most important alternative market for Android apps.

According to AndroZoo creators [ABKLT16], the primary constraint in the app collection is the download of applications from Google Play, because of an account level limit. For this reason, we separate the application information gathering step from the actual download, performed once the analysis is required (Section 4.1).

Given the difficulties encountered by previous studies with the dynamic analysis, we propose a metric to avoid wasting too much time and decide when necessary to perform it (Section 4.3).

Beyond the study on application security, one of the main objectives is the creation of a dataset that remains updated with respect to the application stores, i.e., Google Play Store and Aptoide. Indeed, to overcome the limitation of existing datasets that become outdated quite fast, the proposed architecture needs to include an automatic update of the information for every new app release.

For each parsed application, we are interested in recording the package name of an app, its title, icon, category, developer, number of installations, privacy policy, the number of ratings and average score, the presence of advertising, and the timestamp of the last update. Each time an app is analyzed, we also include the origin market, the application version, the hash of the file, and the date of analysis. The user should be able to filter this information using a search bar and sort the obtained results by an attribute.

3.1.2 Security Analysis

In the rest of this Section, we define which are the security measures to include, explain the reason why we consider important each one and show how the user can retrieve this information from the platform.

¹<https://www.aptoide.com>

Among the information obtained through the static analysis (detailed in Section 2.2), we are interested in evaluating:

- the structure of the app (i.e., activities, services, providers and receivers),
- the usage and request of permissions;
- the used APIs,
- the presence of security vulnerabilities (e.g., insecure webview JS enabled);
- the usage of third-party libraries or frameworks;
- the level of obfuscation;
- the presence of malicious code;

Many studies [BBD16, Der18, LWW⁺18] show how third-party libraries are dangerous for Android applications that are not constantly updated, although the Google ASI program² [DBF⁺17] reports the presence of security issues. Indeed, once a vulnerability in a library is discovered, numerous apps could be affected. Therefore, our architecture should enable the detection of all applications that are using a vulnerable component: this is useful both for researchers as a starting point to find a bug or to implement a notification system for developers. To achieve such a goal, the platform should find and collect all the libraries used by an application and allow the user to search the package name of a specific one. Furthermore, the architecture should also report the libraries related to tracking or advertising for privacy analysis purposes.

Since most applications communicate over the Internet, the security of the connection is an essential security measure. Typically, the security evaluation of the network connection is performed during dynamic analysis, although static vulnerability analysis modules could also detect insecure requests or configurations. Connections that do not use SSL are the main problem, as traffic is not encrypted and could be intercepted. A wrong configuration of the protocol is also a problem: the certificate chain may not be checked, thus accepting all the certificates, including that self-signed or expired. To minimize Man-in-the-Middle attacks, Android recommends to i) avoid any plain connection, and ii) enforce apps connection with certification pinning techniques.

Information about the reached internet domains is also relevant both to study the most contacted endpoints and to look for their security reputation. Indeed, researchers may find a host involved in a privacy leak and query the architecture to discover if also other applications communicate with the same insecure endpoint.

²<https://developer.android.com/google/play/asi.html>

Therefore, the architecture should be able to track all the accessed URLs with the corresponding network security configurations.

The dynamic analysis should also provide other information such as i) the accesses of the app to internal and external memory files, ii) the leaks of personal or device data, and iii) the usage of hard-coded keys, password, or private tokens.

Besides the filtering and sorting features, the system should also provide general security statistics like the most used libraries and contacted domains or the most requested files and leaks found at the same location reported for more applications.

3.2 Requirements

3.2.1 Table View

To formalize the thesis goals, we organize the information described above in functional requirements (FUNR), security measures (SECM), and statistics (STAT). The first group indicates the most important operations that the platform should allow to perform, the second one includes the significant results of the analysis, and the last one collects the global metrics to show.

Table 3.1 presents the full list of requirements and their corresponding IDs. In the upcoming chapters, we refer to a specific requirement by using its ID.

ID	Description
FUNR1	Filter applications using the package name or a keyword in the title.
FUNR2	Filter applications from a specific developer or category.
FUNR3	Sort applications by the number of installations or their score.
FUNR4	Filter apps that do not include a privacy policy on the Play Store page.
FUNR5	Filter applications that requires a specific permission.
FUNR6	Filter application that use a certain category of APIs.
FUNR7	Filter applications that include a specific library inserting its package name.
FUNR8	Filter applications that use a low obfuscation level.
FUNR9	Filter applications that contact a specific domain inserting its address.
FUNR10	Filter applications that use insecure connections.
FUNR11	Filter applications that access to a specific file path.
FUNR12	Filter applications for which a leak is reported or an hard-coded key found.

SECM1	The list of components included in an application
SECM2	The list of permissions required by an application.
SECM3	The list of APIs used by an application.
SECM4	The list of libraries included in an application.
SECM5	The request to read the Device ID.
SECM6	The list of vulnerabilities reported for an application.
SECM7	The obfuscation level for an application.
SECM8	The probability for an application to be a malware.
SECM9	The list of hosts contacted by an application with network security settings.
SECM10	The list of files in the external memory used by an application.
SECM11	The list of leaks found for an application.
SECM12	The list of hard-coded keys reported for an application.
STAT1	The most required permissions.
STAT2	The most used APIs.
STAT3	The most imported libraries.
STAT4	The most common vulnerabilities.
STAT5	The most contacted domains.
STAT6	The domains most involved in a leak.
STAT7	The most used files.
STAT8	The files most involved in a leak.

Table 3.1: Platform Requirements

3.2.2 Additional Studies

Besides the implementation of these feature, one of the purpose of the thesis is also to provide the results of the queries in the most interesting cases (Section 6.2.1), statistics output (Section 6.3) and produce additional information not directly shown in the platform (Section 6.4), such as:

- Study differences between versions of an application.
- Study differences for the same application between different stores.
- Study any relation between the used permissions and the reported vulnerabilities.

Furthermore, we would like to retrieve from the last point if it is possible to report a statistical security risk classification based on the information accessible to the user when an app is installed.

Chapter 4

Design

In this chapter we propose a solution to the previously set objectives, first by defining a design model and then reporting details about the architecture of each feature.

4.1 Architecture

To meet the requirements defined above, we propose an architecture able to retrieve application information, analyze its security and manage the obtained reports and statistics. This architecture is called Appregator.

Appregator is made up of four components: Crawler, Worker, Database and Server. The database allows coordinating the other elements, each of which will communicate directly with it. The crawler takes care of looking for new applications, while the worker downloads and analyzes them. The server exposes an API accessible to the clients through a GUI that allows users to extract and filter results interesting for security.

4.1.1 Workflow

The crawler is a process always running that saves Play Store information: when it finds a new application or an update for an existing one, using the *updated* field timestamp returned by the market, the app is stored in a local object with the attribute *download=true*. Once the collection cycle is completed, the program stores the local list in the database: the worker will perform a transaction reading the most popular apps by choosing the elements with the most installations and ratings that verifies the *download* flag, then it sets the condition to *false* and eventually increases the *updated* date.

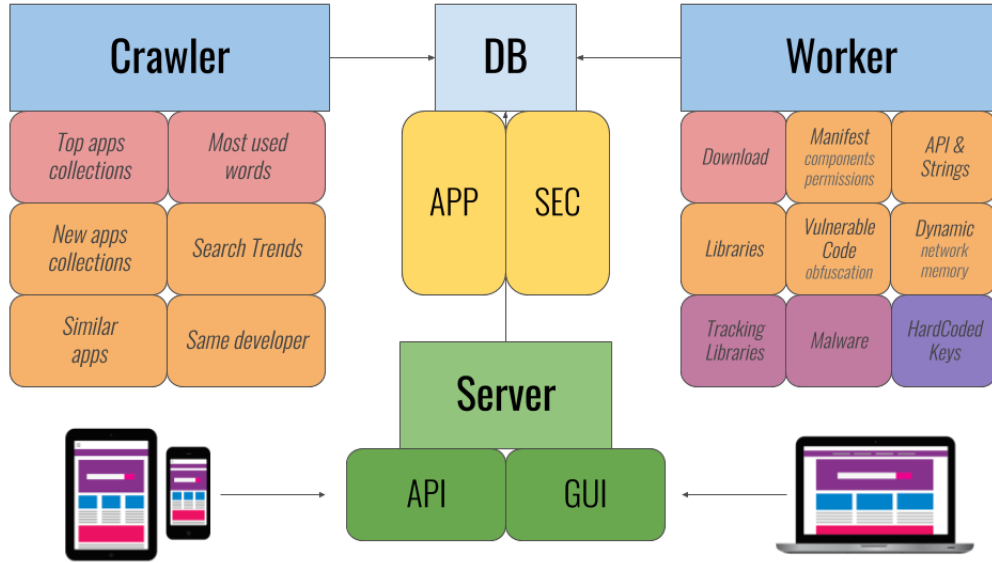


Figure 4.1: Aggregator architecture

While there may be multiple workers, there is only one crawler and a centralized database. A typical deployment could consist of some slave nodes with one worker and a master node on which are installed the database, the crawler and the server: the latter components can run on the cloud since the only requirement is enough space, but the analysis requires a lot of resources and therefore a bare-metal server is needed.

4.2 Crawler

The market proposes different collections of applications and each application belongs to a category: when the crawler is started, it considers free items that appear in each category of the store for the collections TOP FREE and GROSSING of apps and games. After that, it runs an infinite loop examining TRENDING and NEW FREE applications and scraping apps from the same developer or similar to those already saved in the current step, thanks to the suggestions provided by Google Play.

The Crawler looks for apps also thanks to a search feature: on the first run it uses an input of common terms of different languages, and then it searches trending words of the last 10 days, obtained through Google Trends¹ service. Starting from each word, multiple queries are considered using the suggestions provided by the auto-complete feature of the market search bar and each one returns different results to compare with the values already saved.

¹<https://trends.google.it/trends>

Keeping a list of already downloaded apps in memory may run out the space available to the process (Section 5.1.3). The constant growth of crawled applications and the goal of managing a study on a large scale oblige Appregator to limit the size of such a list, even if the Crawler stores information only about the date of the last update.

However, keeping the information in the main memory only of some applications implies saving apps that may have already been analyzed and therefore relax the consistency condition with the database. Since the priority is to save the worker time, it is necessary to avoid repeating an analysis that has already been carried out, hence the duplicate of the primary key cannot be managed simply with a replacement as the *download=false* flag would become *true*. These entries cannot be ignored because information on new updates would be lost and therefore we have to consider a more complex model, faster than scanning the list of failed insertions and compare the *updated* field for each element. To correctly manage the exception, it is indeed necessary to check if a new update is available only for the already analyzed apps and replace those still to download (Figure 4.2).

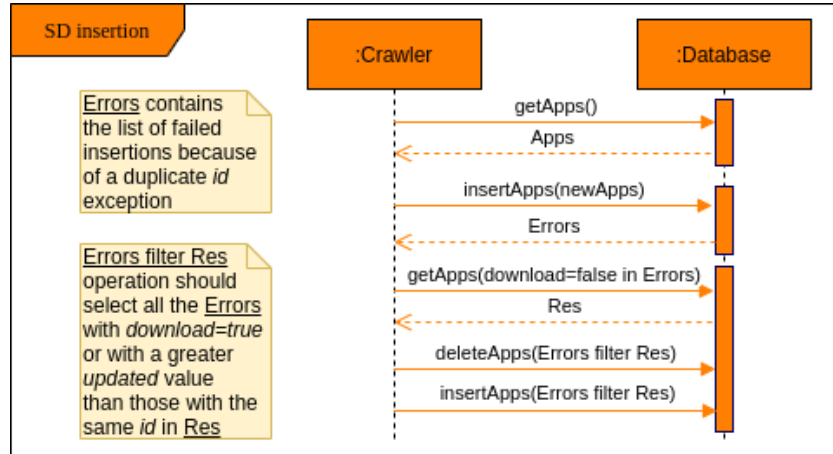


Figure 4.2: New applications insertion sequence diagram

4.3 Worker

The Worker gets N prioritized apps to download and analyze, it should also provide a partial rollback if some tasks fail. This is the most time-consuming job, both for the limit in the package downloads imposed by markets, Google servers are especially restrictive, and the resources necessary for the analysis. We organized the security measures of the previous chapter in units (Figure 4.1), following the automatic analysis process described in the background knowledge (Section 2.2): from the manifest, we retrieve information about app components and permissions (SECM1, SECM2), SECM3 goal is obtained through a module able to find strings and used APIs, for which also a category should be provided.

SECM4 has a dedicated component, but we defined an additional one to find tracking libraries. The vulnerable code analyzer is responsible for SECM6 and SECM7. Also, the Worker includes a virus scan (SECM8) and several dynamic analysis modules to cope with SECM9, SECM10, and SECM11. The Device ID access (SECM5) can be retrieved combining the code analysis with the dynamic one, while another module tries to identify the usage of hard-coded keys (SECM12).

The Worker should also check if the last version of the downloaded files from the Play Store is the same in the Aptoide market: it has a documented public API for the search feature², we use it looking for the package name of an app and limiting the results to one. Being a version of an application denoted by a name (*verName*) and a code (*verCode*), the Worker compares the SHA-2 of APKs downloaded from the Play store with those that have the same *verName* coming from unofficial markets. If they have a different hash, the Worker performs a new security analysis. Besides, if the apps with the same *verName* but different hash also matches in the *verCode*, the Worker executes the malware analysis, to detect whether those apps are potentially repacked with malware codes.

The most expensive step for the Worker in terms of resources and time is the dynamic analysis, so we propose a metric based on the results of the static API monitor to determine if it is needed. Since it retrieves URLs, contacted domains and leaks via the net, we expect the app to use at least one API in the INTERNET category. Similarly, if it detects a leak to external memory, the app should require FILE access and if the leak concerns device details like a phone number or the device id, a function related to PHONE information should be executed. We expect to detect hard-coded keys thanks to CRYPTO functions calls, then, of course, this kind of category is required. However, file and network access are very common, so at least three of the four categories listed above are needed.

The presence of hard-coded keys is found by monitoring the execution of functions categorized as cryptographic: we compare all the strings extracted from the application with the arguments passed to the call. False positives are common because there are also parameters to configure encryption: we do not classify all the functions involved by selecting the number of the argument containing the key, but we experimentally establish rules to limit errors. The chosen method allows us to save time avoiding to manually search through the documentation of each API and listing the required information, by looking for a general way to reduce the number of matches and guarantee a better future support thanks to the independence from the platform. In particular, we decide to ignore arguments with fewer than four characters from the comparison, as they were too short to be a token and save the most frequent results to build exclusion lists and false-positives patterns.

²<https://co.aptoide.com/webservices/docs/7/apps/search>

4.4 Database

The Database is the most important component of the system since it stores all the information collected and provides data to the server, which allows the user to perform the queries. It maintains two tables: one with information available in the Play Store, identified by the *appId* (package name), and the other with the results of the security analysis, which has a unique *verCode* (version) for each app associated to the source *store*. The conceptual schema of the Database is described in Figure 4.3.

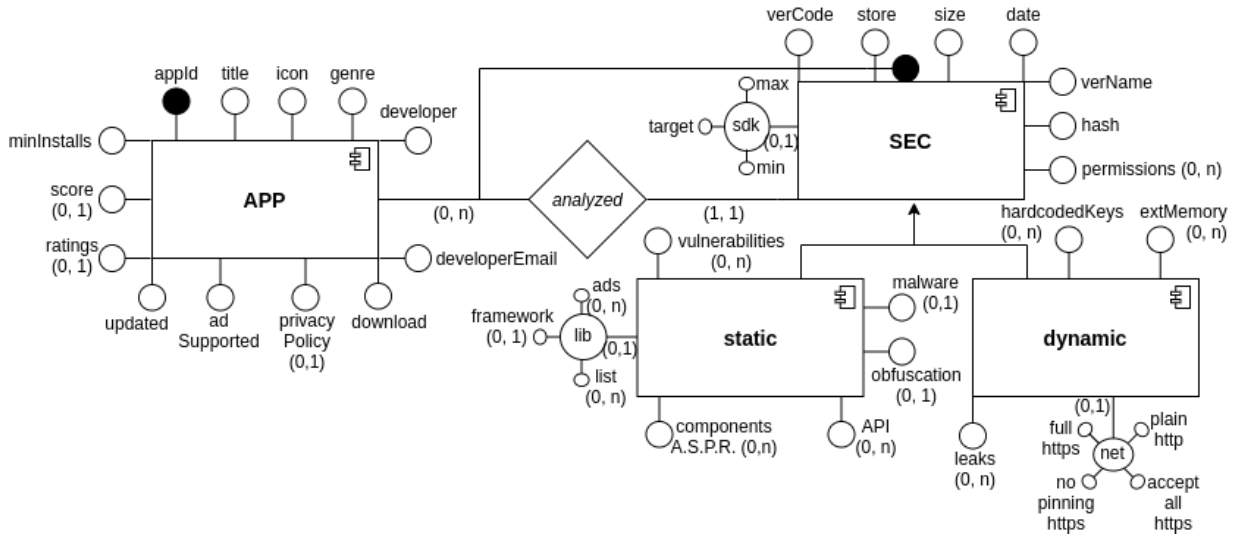


Figure 4.3: Conceptual Schema

4.4.1 Typical Workload

Below, we define the operations that the database should handle most frequently:

- The Crawler writes in the APP table at scheduled time, the frequency of access will be established experimentally.
- The Worker reads and writes (it edits the *download* flag) in the APP table for limited results query, corresponding to the number of apps to analyze in a single step (N).
- The Worker writes in the SEC table every time a group of apks is analyzed, the frequency will probably be of some hours but it will only append new elements.
- The API will often perform read queries in both tables.

4.4.2 Logical Schema

From the analysis of the conceptual schema and the workload, we defined the logical schema: in particular, there is no need to separate the information retrieved by static and dynamic analysis.

APP (appId, title, icon, genre, minInstalls, ratings, download, developer, developerEmail, score, updated, adSupported, privacyPolicy)

SEC (appId^{INFO}, verCode, store, size, date, sdk, verName, hash, permissions, leaks, net, hardcodedKeys, extMemory, lib, malware, obfuscation, components, API, vulnerabilities)

4.5 Server

The Server exposes a Rest API endpoint and a Web Interface that enable the querying of the database with filtering, sorting, and features selection options.

We should define a routing pattern flexible enough to determine the table to be examined by the path of the URL and apply the proper handler, a controller function able to retrieve data from the DB: the MVC architecture seems to satisfy these criteria and it fits perfectly with the required purpose to build a web application.

APPregator

List

SEC

Category = Social

Search

<< < > >>

3647 results









Application	Category	Developer	Installs	Ratings	Score	Ads	Policy
 Facebook com.facebook.katana	Social	Facebook android-support@fb.com	5000M	96M	4.24	true	
 Snapchat com.snapchat.android	Social	Snap Inc snapchat@snap.com	1000M	21M	4.35	true	
 Instagram com.instagram.android	Social	Instagram android-support@instagram.com	1000M	95M	4.46	true	
 Facebook Lite com.facebook.lite	Social	Facebook lite-android-support@fb.com	1000M	13M	4.23	true	

Figure 4.4: Example of Appregator Web Interface

To optimize the user experience and avoid returning large sets, the server adopts operators typically implemented in a query language like count, limit and offset. If the client filters the data, an estimation of the number of results of the query is returned, to allow him to access the output in pages of 25 elements and jump to the next sections using navigation buttons (Figure 4.4). However, it should be possible to download all the filtered results, that could also represent the whole dataset: to avoid storing the object in the main memory, the server should provide a stream from the database to the client.

To differentiate the computational load, two endpoints are needed: one for the generation of navigation links and the other to start the download stream. These must have a general purpose, applicable both for the information on the applications obtained from the Play Store and for those returned by the analysis, but another function related to the latter should deal with returning statistics.

We are interested in finding out what are the most requested permissions (STAT1), used APIs (STAT2), imported libraries (STAT3), common vulnerabilities (STAT4), contacted domains (STAT5) and opened files (STAT7) in general or when a leak is detected (STAT6, STAT8). We may choose to perform this query only after a certain amount of time has passed and hence be more efficient or always executing it and have updated results. Since the analysis step needs time, we do not expect results to change often and therefore the first solution seems more reasonable, but the second would allow us to reuse the filter pattern defined above and retrieve statistics valid for a subset. Indeed when the query is executed on smaller sets it is faster, so we decided to save a cached version of the general results with the number of apps to which is related, updating it when more analysis is available, and perform a new query when results are filtered; this behavior could be generalized to save the most requested queries.

Chapter 5

Implementation

In this chapter, we provide a report about the used technologies, motivate their choices and go deeper into the details of the features from an implementation point of view.

5.1 Technologies

The security analysis is performed through Approver, a SaaS developed by Talos¹ consisting of various micro-services, a description of the main ones is presented in the next Section.

We decided to remove some of the original modules and call the most complex ones only if certain conditions are met, as in the case of the dynamic analysis (Section 4.2). This choice was mandatory for malware recognition as the service used imposes limits according to the price plan preventing a large-scale analysis, moreover, the chances to find malware on the official store with an automatic tool are very low.

The research purpose of the project allows us to include different services that are not compatible with proprietary software such as Approver: for example VirusTotal², another malware analysis module, or the tracking detector Exodus³, which is not very useful for a developer who uses Approver to prevent vulnerable code.

Approver is a web service, but to reduce latency in communications, a local version should be included in the platform: this approach would allow performing only the most interesting steps of the analysis, accessing single operations in a micro-service style and adding functionality to the software.

¹<https://talos-sec.com/>

²<https://www.virustotal.com>

³<https://exodus-privacy.eu.org>

5.1.1 Approver

Many of the components which constitute the worker, as defined in the architecture (Section 4.3), correspond to some of the Approver modules:

- The Decompiler module extracts code and resources from the application: code in *.dex* files is decompiled in *smali* format, the manifest and strings are saved to disk.
- The Manifest Parser module analyzes the manifest file and produce a JSON with permissions, activities, services, version and name of the application.
- The String and API Analysis module analyzes the code looking for strings and categorizes all the Android API calls.
- The Permission Checker module produces a list of permissions, taking into account both those used in the code and those declared in the manifest.
- The Library Detector module looks for known third-party libraries inside the app.
- The Vulnerability Checker module identifies code pattern considered a security risk and label its severity.
- The Dynamic Analysis module simulates user input and it saves a log of called APIs, filesystem interactions and network traffic checking if SSL is used and if pinning or certificate chain is enabled. It also looks for leaks checking if the phone number, the device ID or the email address of the user is saved into a file or transmitted via net.

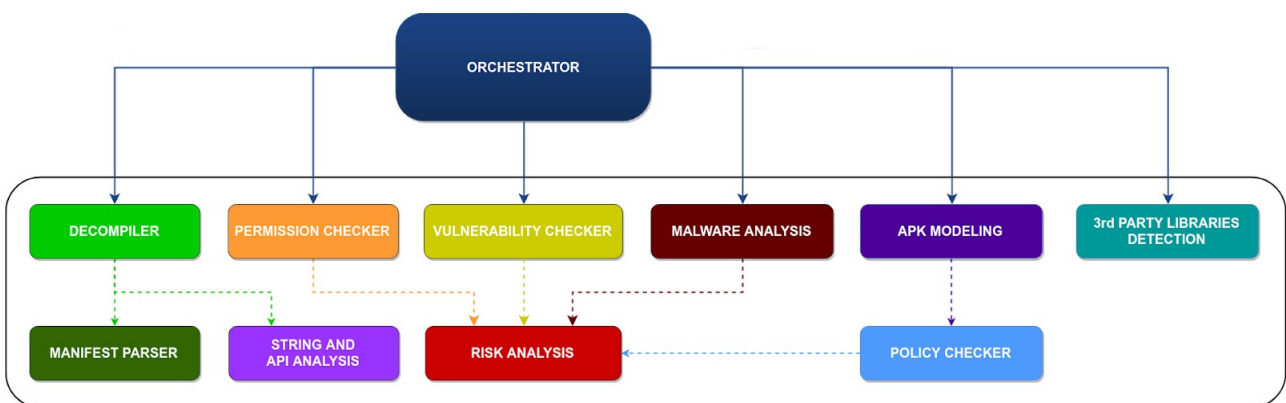


Figure 5.1: Approver Static Modules

When an app is submitted to Approver, it will start all available analysis: to perform the dynamic tests only on a subset of applications, we have to edit its source code.

The responsible for the upload is the APK Storage, which will make the file available for the other modules. Once the app is ready, the Orchestrator module will manage the analysis cycle, collecting the outcome of each test. If we try to upload an already analyzed application, the APK Storage responds with the status code 409, but it still sends a callback to the other modules, that will provide cached results. Therefore when we need to study the dynamic behavior, we should not worry about the repetition of the static tests, but we can simply add a field in the upload request that specifies which kind of analysis is required.

Modules are written using Python Flask technology and the Spring framework in Java, we defined a *flow* field in the controller of the APK Storage server and managed the request in the Orchestrator checking if the flag is provided or calling every modules if it is not, ensuring back-compatibility as default. We defined the type of analysis in the MySQL database: this could be useful also to provide different scan plans in Approver.

5.1.2 MongoDB

The choice of the Database Management System is the most important since the purpose of the platform is comparing android apps data. Despite the domain of the application does not require a high level of consistency, we are not really interested in replicas because we don't expect a huge number of users working on the platform since it's intended for researchers: rather large space and a clever way to access information will be required. Choosing a document-based solution like MongoDB should allow us to query nested values efficiently, useful to analyze results, thanks to multikey indexes and it has a relaxed schema who guarantees that the data does not become useless if new features are added to Approver; a backup service must also be provided.

5.1.3 NodeJS and Docker

To extract application data from the Google Play Store we rely on the npm library `google-play-scraper`⁴. There are similar alternatives in different languages, but we chose this one because it has an active community, capable of keeping up with changes in the market API. As a consequence, we used Nodejs, in which the library was written: to be consistent with this choice, the use of the framework Express for the HTTP server is the most obvious option and it can be combined with HTML and CSS pages for the front-end development.

⁴<https://github.com/facundoolano/google-play-scraper>

Since the worker must update the release date for the analyzed version, it uses the library mentioned above. Adopting the same language allows us to reuse some modules, such as the one that communicates with the database. However, it also calls python scripts to download the applications and start Exodus privacy analysis: in order to provide portability and simply deploy the platform without worrying about dependencies we use Docker. It has the advantage to allow to set the restart of the containers on failure or when the job is finished and it has resource allocation mechanisms useful to limit the memory usage. This is essential to manage the crawler resources: we monitor the container to empirically determine during the execution cycle when to send the information to the database or perform a local backup of the data not yet saved (volumes persist on containers reboot).

We should also take care of NodeJS memory limits: JavaScript objects are stored in the V8's manager heap, whose dimensions are set with the flag `-max-old-space-size`. Older versions of NodeJS have this limit set at 512 MB, but on modern machines the last version uses 2 GB. We wrote a logging mechanism that saves the dimension of the active memory of the process (the portion of allocated resources actually used) to choose a value that allows keeping information about a big number of applications.

The amounts of apps to retrieve from the database depends on the above configuration: since this list is an object with the `appId` as key and the access to a field in JavaScript has complexity $O(1)$ ⁵, a drop in performance should not occur when the crawler checks if the app has already been saved also if many entries are considered. Taking into account this consideration and comparing the results shown by the log, we decided to limit the query to half a million of results and set the flag to 4GB.

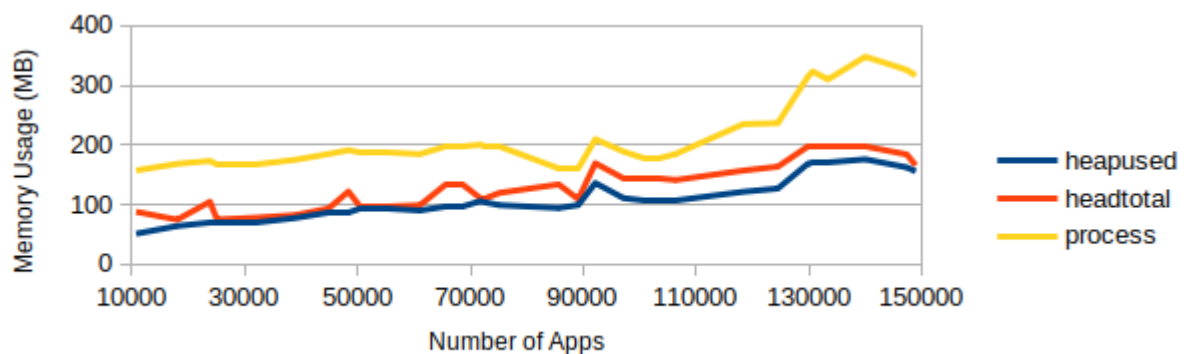


Figure 5.2: Process memory usage collecting [10K, 150K] applications

⁵<https://stackoverflow.com/questions/12241676/javascript-objects-as-hashes-is-the-complexity-greater-than-o1/12255071#12255071>

If we set docker memory limit according to the chosen value, when too much resources are used we expect the container to be stopped, but this may not happen according to *Node.js memory management in container environments* article from IBM Developer [Yat19]. Indeed we performed some experiments limiting these resources and noticed how the OOM-KILLER (the kernel mechanism that manages out of memory errors) does not actually come into action, but since the container entry point coincide with the node process, it is restarted anyway.

A similar issue affects some of the Approver modules: since they are not meant to manage a large scale analysis, a big request of resources and an intensive usage could cause the crash of one secondary process in a service, without restarting the container. To avoid this situation we could perform the analysis on groups of applications and when one group ends, Approver should be restarted. This approach has the advantage that once an analysis is completed, we can avoid sending a delete request for each app, which is propagated internally to Approver to each module, and just remove the volumes; furthermore if a single module does not send a response from a certain period of time, it could be restarted.

In order to obtain this behavior the worker, who runs inside a docker container, should be able to stop Approver containers that run on the same host, but this feature represent a critical issue from the security point of view of the system. There are two solutions, both not without risk: the host exposes the Docker Engine API to receive HTTP requests from the container to interact with the Docker daemon or we directly mount the docker socket of the host system into the container. In the first case we have to set up a communication that will take place on the external network using the host IP address or use the docker0 interface and modify the iptables rules⁶, allowing containers to access all the ports. To use the second method we need the Docker CLI installed inside the container: this practice is known as *Docker-in-Docker* and it originates from low-level technical problems that raise with encapsulated Docker instances, usually solved renouncing to independent installations [Wei18].

Security problems emerge because docker commands need root permissions and also if they are executed with the above methods, it is possible to mount the host root file system into a container as a volume and perform a privilege escalation [Cha19]. To limit this risk we define a new container that has the only purpose to manage Docker, reducing the probability to expose a vulnerability because it has far fewer installed software, libraries and implemented features. It should share a bridge network with the worker and expose a port to listen for restart requests, received data is never inserted into shell commands but only used to decide which is the action to execute. Note how similarly also cloud services⁷ do not allow to directly interact with the demon, implementing a separate API.

⁶<https://stackoverflow.com/questions/31324981/how-to-access-host-port-from-docker-container>

⁷<https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html>

5.2 Technical Details

5.2.1 Crawler and Worker

When too many requests are made to the Play Store, the server starts responding with the status 503 and sending captchas: it temporarily bans the requesting IP if the client does not provide any response. To avoid this situation, the crawler sets a limit to the number of requests to attempt per second, a random value less than 10 for each step of the procedure, and it eventually suspends the process for two hours. It is not enough to check the HTTP response code: the market could also cause the request to time out, respond with an internal server error or reply that the page does not exist. These conditions often occur before the captcha is returned, therefore an error counter is kept and we set a threshold beyond which requests must be interrupted.

It may also happen that not all the information for an application is returned, in this case, the resource is requested several times. After some attempts, we conclude that the field is not present if optional, like the privacy policy, otherwise we add the app to a list to resubmit later.

Since we must be sure to download apps from the Play Store we can not rely on third parties APK collections, therefore there are two options: install them on a virtual machine and access `/data/app` folder, where `.apk` files are stored or try to interact directly with the store API. We chose the second approach because it is more efficient and we started from a python script⁸ that takes as input a package name and it launches the download. In order to work, it should be configured with google credentials and the Google Service Framework ID of a device, generated when the first app is downloaded in Android from Google Play. It has some limitations because it simulates a system running with the SDK 23 and requests are associated with the language and available features of the device (like the presence of NFC technology).

To support newer versions of the operating system, we have to edit the library imported by the script replacing the user agent string of the request with the one used by the play store application in the required Android SDK. However, the header of the request is encrypted and capturing clear HTTPS traffic starting from Android Nougat is not trivial since the device does not trust a user installed certificate unless it is not declared inside the application and moreover it is still not accepted if the app performs SSL pinning, as the Play Store application does. Usually, the only alternative is instrumenting the application and try to bypass the pinning, which could be extremely difficult in apps that implement anti-reversing defenses as play services (which are proprietary software, unlike the open source operative system) probably do.

⁸<https://github.com/ClaudiuGeorgiu/PlaystoreDownloader>

Another option could be to edit the user agent string of the older android version, trying to guess which information is represented by each field and replace it with the updated one. Actually most of the values can be found connecting the ADB shell to Android⁹ or using collections of hardware device properties¹⁰: after some attempts mixing information obtained with these two methods, we were able to produce a header working for both the SDK 26 and 28 (Code-Snippet 1).

```
if self.sdk_version < 26:
    downloadManager['User-Agent'] = 'AndroidDownloadManager/4.1.1'
    + '(Linux; U; Android 4.1.1; Nexus S Build/JR003E)'
    playstoreFinsky['User-Agent'] = 'Android-Finsky/4.4.3'
    + '(api=3,versionCode=8016014,sdk=' + self.sdk_version
    + ',device=hammerhead,hardware=hammerhead,product=hammerhead)'
else:
    downloadManager['User-Agent'] = 'AndroidDownloadManager/8.0.0'
    + '(Linux; U; Android 8.0.0; STF-L09 Build/HUAWEISTF-L09)'
    playstoreFinsky['User-Agent'] = 'Android-Finsky/8.5.39'
    + '(api=3,versionCode=80853900,sdk=' + self.sdk_version
    + ',device=crackling,hardware=qcom,product=crackling)'
```

Code-Snippet 1: Comparison between the original and obtained user agent

When the worker starts it tries to download 32 apps to submit to the analysis step, launching four simultaneous connections every 40 seconds (times interval are chose empirically to maximize successful responses). For compatibility with the dynamic analysis virtual machine, we try to get apps simulating Android 6.0 as first choice and then we download those who failed using Android 8.1 and 9: the ID used for the last version is the only one associated with a physical device (in the Italian language).

If an app is not obtained after this three steps, we set its *download* field in the database to *true* only if a network-related error has occurred to avoid to have high priority items not compatible.

Once the analysis for a group of applications is launched, the worker starts sending periodic requests to Approver Orchestrator module to get the status of the analysis and retrieving results of those that have finished.

⁹<https://blog.onyxbits.de/how-to-get-the-google-play-user-agent-for-a-given-device-140/>

¹⁰<https://github.com/yeriomin/play-store-api/tree/master/src/main/resources>

When all the analysis are completed or a maximum time limit reached, the results can be stored in the database and if an environment variable with the path where to save the apks is provided, the files are copied before removing the container volumes. We experimentally set the time limit estimating ten minutes for the analysis of a single application and multiplying it for the number of successfully uploaded apps to Approver, adding five minutes for each requested dynamic analysis.

5.2.2 Database and Server

Elements with the same hash in the database will refer to the same analysis: this choice should also reflect on the GUI. When the user select the details for an application, if the *store=aptoide* field is recognized, it should check if the playstore version has the same hash.

In order to generate the statistics, we provide a pattern to apply for each query. Since analysis results are often organized in nested documents, we name KEY the item which contains the array of objects we are interested in and VALUE the field with the information to extract.

Below the steps to get the N most frequent values for KEY[VALUE]:

- Filtering using the client starting query criteria
- Use Mongo unwind operation on KEY to deconstruct the array and obtain a document for each element
- Group by VALUE and name COUNT the number of elements in the same group
- Match elements with COUNT > 1
- Sort by COUNT in descending order
- Limit the query to N results

For example, to retrieve the 10 most common vulnerabilities, we chose the corresponding attribute as KEY and the ID of the vulnerability as VALUE. If we want low priority problems to be excluded, we can add an optional filter after the unwind operation that depends on the kind of information we need, like $\{ \$match: \{ "vulnerabilities.level": \{ \$ne: "Low" \} \} \}$. The KEY could also be an embedded document, but since often there are different arrays at the same level as for the declared and used permissions, we can change it using an object which associates to a field its subdocuments. In this case we can add a last step executed by the server to retrieve general results about the arrays, extracting the N highest values of COUNT.

To filter the results with the rules provided by the user, we perform a client-side parsing to allow the usage of simplified key names to insert in the search bar and suggested by an auto-complete feature, ensuring a relaxed schema for spaces, uppercase letters, strings in quotation marks or lists of values. The obtained query is inserted as URL parameters in a GET request to the server and then used to achieve the criteria object to pass to mongo thanks the *query-to-mongo* library¹¹. Using directly the API, suggestions and keys control are not provided and therefore, values must be exact to get a response. However, more operations are allowed, like sorting by more keys or select which fields return.

Since the user can specify limit or sort operations, we can not just sanitize the whole string, but we should perform an additional server-side parsing phase in order to resist to NoSQL injection attacks [Coo19]. We implemented this feature as good programming practice, but there is actually no information in the database that the user should not access. The client is not expected to perform writes, and therefore the web server has only read access to the database. Sort operations are not permitted on too large collections, and therefore we limit the query unless an index is specified: we provide indexes on the number of installations for the APP table and on vulnerabilities and libraries for the SEC one.

¹¹<https://github.com/pbatey/query-to-mongo>

Chapter 6

Experiments

In this chapter, we present the most interesting privacy and security related queries to test the requirements of the platform, then we show the computed statistics and discuss some more advanced use cases.

6.1 Experimental Setup

To test Appregator, we installed it on single machine with a quadcore 3.40 GHz processor, a SSD with 256 GB of storage and 24GB of RAM. Using this setup, we collected in one month 200.000 apps information from the market and analyzed 3500 different application versions from the considered stores.

6.2 Query Capabilities

The functional requirements and security measures defined and implemented in the previous chapters, are evaluated in this section thanks to the search feature of Appregator.

6.2.1 Privacy

Using the Web Interface, the user can filter the apps that do not contain a field just using the search bar. For example, if the operator wants to filter apps that do not include a privacy policy on the Play Store page (FUNR4), he can execute the search *!policy*. If he also wants to find the subset which declared to include advertisements, he can append a semicolon with the new command *ads=true*, as shown in Code-Snippet 2.

```
!policy; ads=true
```

Code-Snippet 2: Query to find apps with advertising and without a privacy policy

By using Code-Snippet 2 query on the entire dataset, Appregator reports that almost 13.000 applications do not include a privacy policy in the Play Store page on the last check, and about 7.000 of these use advertising anyway.

Furthermore, we can sort them by the number of installations by clicking on the table column name (FUNR3): four apps have at least 100 or 50 million users (Figure 6.1), 35 apps have more than 10 million installations, 350 more than 1 million installations, and more than 1.500 have at least 100.000 users.





	Application	Category	Developer	Installs	Ratings	Score	Ads	Policy
	TubeMote com.tubemote.app	Entertainment	TubeMote contact@tubemote.com	100M	419565	3.43	true	
	Chess com.jetstartgames.chess	Board	Chess Prince help.chess@mail.ru	100M	1M	4.33	true	
	Checkers com.dimcoms.checkers	Board	English Checkers help.checkers@mail.ru	50M	388279	4.32	true	
	Escaping the Prison air.com.puffballsunited.escaping...	Casual	PuffballsUnited puffballsunited@gmail.com	50M	471042	3.89	true	

Figure 6.1: Top 4 of most installed apps without a policy on the Google Play Store

By selecting one application or going to the security page, the platform shows the list of analyzed app versions and offers the possibility to inspect the details for a specific analysis. A search bar is also provided in this case: the operator can look for apps involved in a leak (FUNR12) by using the *leaks* keyword in the search bar.

By using this query on the dataset of analyzed apps, we discovered 21 apps with a potential leak, 17 of which come from the Play Store. In detail, 12 privacy leaks are saved to file, 12 exposed on the net, 19 refer to the device ID, and 5 to the user email.

Since *Android best practices for unique identifiers*¹ suggests developers to avoid using hardware identifiers like the device ID and use GUIDs to identify app instances uniquely, we inspected the collected dataset. Figure 6.2 presents the results obtained by the query in the Code-Snippet 3: 9 Google Play Store apps send the Device ID to a server.

¹<https://developer.android.com/training/articles/user-data-ids>

```
leaks.net.category=device; store=play
```

Code-Snippet 3: Query to find Device ID leaks via net in the Play Store

Such an issue is particularly relevant when using advertisement libraries as a user-resettable identifier needs to be used. Despite this, the authors of *Ad IDs Behaving Badly* [Ege19] reported to Google how ads libraries transmit over the net the device ID instead of a user-generated Advertising ID. The results obtained by Appregator confirm, one year later, the worries of the report, thereby also proving the usefulness of the website search feature (FUNR9) since some of the domains involved are the same reported in the study.

It is also possible to find the usage of the device ID from the static analysis (SECM5) with *vulnerabilities.vulnerability_id=SENSITIVE_SECURE_ANDROID_ID* and get more than 1500 results. Still, this result needs to be validated through the dynamic analysis phase since from Android 8 the device ID is different for each app-signing key and, in some cases, it could be used properly on these devices.

	Application	verName	verCode	Store	Hash	Size	Date
i	com.junerking.archery	3.1	20	play	1b357561a3bb...	23MB	16/2/2020, 07:40
i	com.neuralprisma	3.2.4.413	7000413	play	e0e7dec8bacb...	11MB	17/2/2020, 15:43
i	paint.by.number.pixel.art.colori...	2.12.1	1020	play	5771f49e796d...	36MB	5/3/2020, 12:07
i	com.gamebasics.osm	3.4.51.5	345105	play	0b58546fed86...	11MB	6/3/2020, 19:25
i	com.yahoo.mobile.client.android....	1.20.3	91596454	play	b2414799b92c...	30MB	8/3/2020, 07:04
i	com.hypermedia.songflip	1.1.10	1803131913	play	5d4c704f932c...	9MB	9/3/2020, 10:09
i	com.pinssible.fancykey	4.7	4146	play	431eb17ff9e8...	19MB	9/3/2020, 19:30
i	com.trulia.android	11.8.1	719	play	9b9c16c8455e...	13MB	11/3/2020, 21:03
i	com.chatous.chatous	3.9.87	379	play	216c107efb72...	46MB	11/3/2020, 21:03

Figure 6.2: Play Store apps that send the Device ID to a server

The query *vulnerabilities.vulnerability_id=SENSITIVE_DEVICE_ID* returns that 983 Play Store applications access the IMEI number. Furthermore, since such reading requires the permission *READ_PHONE_STATE*, we fine-tuned the query as in Code-Snippet 4 and obtained 351 results.

```
store=play;
vulnerabilities.vulnerability_id=SENSITIVE_DEVICE_ID;
permissions.declared=android.permission.READ_PHONE_STATE
```

Code-Snippet 4: Query to find the IMEI in the Play Store

Finally, we can use the download feature to search the libraries that save this information counting the most common Java classes writing a simple script (Code-Snippet 5): all the obtained results are related to advertising and listed in Table 6.1. The download feature eases the offline analysis, indeed the server provides operations for the most frequent use cases only.

JAVA CLASS	LIBRARY	APPS
Lcom/umeng/commonsdk/statistics/common/DeviceConfig;	Umeng	71
Lcom/unity3d/services/core/api/DeviceInfo;	Unity3d Ads	45
Lcom/tapjoy/TapjoyConnectCore;	Tapjoy	16

Table 6.1: Advertising libraries that read the IMEI code

```
import json
from collections import Counter

dict={}
with open('imei.json') as json_file:
    data = json.load(json_file)
    for app in data:
        for vulnerability in app["vulnerabilities"]:
            if vulnerability["vulnerability_id"] == "SENSITIVE_DEVICE_ID":
                for code in vulnerability["vulnerable_code"]:
                    if code["class"] not in dict:
                        dict[code["class"]] = 0
                    else:
                        dict[code["class"]] += 1
print(Counter(dict).most_common(3))
```

Code-Snippet 5: Python script to count the 3 most frequent classes which read the IMEI

6.2.2 Security

In this part of the experimental evaluation, we tested some of the capabilities offered by Appregator to evaluate the capacity to study the security of apps.

Insecure Connections. To find the usage of insecure connections (FUNR10), the security operator can execute the query *net.plain-http*. By using the collected dataset, the query reports seven results. Contacting a host without using encryption is a problem if sensitive data is transmitted since the communication enables attackers to sniff and inspect its content, and therefore we investigated such results. Among the vulnerable apps, six of them come from the Play Store and have between 10 and 100 million installations.

In the first case, the application is a game and it is also involved in the leak of the device ID, that is transmitted using the insecure connection to its developer website. Another affected application is a web browser and reading its description in the store, we noticed that one of the features offered is the advertising block during the navigation. However, the application itself contains ads and tracking libraries, which is a behavior far from a privacy-oriented adblocker.

Studying connections in a browser app is especially tricky due to the strong dependency on the user input for the URL browsing. However, also the static code analysis reports the presence of URLs not in HTTPS (Figure 6.3) and the same host is listed in a Java class called *IWUPClientProxy*. In detail, the network analysis identifies that two parameters named *encrypt* and *qbkey* are inserted in the query string of the link, and this could be a vulnerability because of the clear-text traffic. The connection is related to the usage of the *Tencent Login* library, indeed Appregator also collects the reference to which part of code is involved to facilitate researchers to discover security issues.

The contacted domain is also sent using another insecure connection to a different IP as GET request: this behavior is often related to user data exchange between partner companies in online browsing like cookie syncing², but the cloud service owner of the address (always Tencent) offers an HTTP DNS service whose documentation is similar to this example. This protocol is often implemented without encryption, but using a feature different from the standard settings of the system, could allow MITM attacks if authentication is not provided³.

The app also access to external memory at *WhatsApp/Media/.Statuses*, where picture and video status of the user contacts are stored: however, testing the application we can see that it is an intended-behavior since the feature can be requested from the user.

²<https://freedom-to-tinker.com/2014/08/07/the-hidden-perils-of-cookie-syncing/>

³<https://security.stackexchange.com/questions/155180/dns-mitm-attack>



Figure 6.3: Aggregator security analysis details

Two of the other apps found contact another IP address, associated with a crash report system of the same company just mentioned, and the remaining ones perform the insecure requests to download advertising banners or exchange data with partners.

The filter in the Code-Snippet 6 reports that more than 1400 apps possibly use insecure connections according to the static analysis, but the result is probably overestimated because not all the libraries code is executed and an encrypted connection could be established after an HTTPS redirection.

```
vulnerabilities.vulnerability_id = SSL_URLS_NOT_IN_HTTPS; store=play
```

Code-Snippet 6: Query to find clear traffic use in the code

Vulnerable Libraries. Another operation that we tested regards finding the usage of a vulnerable library just by inserting its package name (FUNR7) or searching its common name. For instance, 50 apps of the dataset include *Tencent Login*, but the version described above is used only by the web browser app.

Unfortunately, Aggregator does not always provide information about the library version, due to the current limitation of the library recognition technology. Indeed, library recognition is not precise due to obfuscation techniques or the lack of libraries signatures. To mitigate such an issue, some additional filter operations should be performed and to help in this process, the permissions related to the methods used in the instance of the library for a specific app are saved.

Malware. The malware field is a probability obtained by adding up the number of positive responses between different antivirus services and dividing the result for the total number of scans. In our dataset, we only find five occurrences with *malware*>0 ranging from 0.016 to 0.032, i.e., a minimal probability.

By using the same approach of Meng et al. [MXS⁺17] that allows identifying repackaging of an app by checking the developer certificate w.r.t. the one in the Play Store, we identify an application between the possible malware with a different value of the certificate. This information is included in the vulnerabilities field with the flag *level=Info*.

Looking for the application hash on Virus Total, we can retrieve the analysis that reports an *Android.PUA.DebugKey* issue, which indicates that the app has a debug certificate: the Play Store would not have allowed the upload, which is instead possible on Aptoide.

However, the differences in the code analysis with the Play Store version of the application are all related to third parties libraries and the alternative market tags the app with *advertising=false*: it looks like a patched a version of the application with ads removed, the other results are probably false positives.

Hard-coded Keys. Appregator reports 8 apps that contain hard-coded keys (FUNR12): as explained in Section 4.3, we insert a check in the code to exclude configuration parameters and manually remove the most frequent values like *HmacSHA1* or *HmacSHA256*.

The platform collects the string, the method, and the number of the parameter: the most common method is *javax.crypto.spec.SecretKeySpec*, which is used to construct a Secret key starting from the string.

This method should be used to store keys generated at run time as authentication tokens, but these strings are inserted into the code and therefore, could represent a security vulnerability depending on the specific case.

Usually, the main problem is the usage of API keys for third-party services, that are saved with this method, to be used later in the code. The best practice is to perform the requests server-side, but also string obfuscation is often used.

We also get some results that are not token, related to the method *javax.crypto.Cipher.doFinal*: strings like *campaign_id* or *partner_name* (which looks related to advertising) are concatenated, but we do not classify them as false positive since are encrypted values included as plain-text strings.

6.3 Statistics

In this section, we present and discuss the results of the computed statistics: when no additional information is provided, it will refer to all the application versions. Note that values with the same hash are considered only once.

The most used APIs (STAT2) are organized in categories: in Table 6.2, we report each method with its category and the relative number of apps. The API analysis is available for 2166 application versions.

API NAME	CATEGORY	APPS
HTTP Requests, Connections and Sessions	INTERNET	2136
WebView GET Request	INTERNET	2031
HTTPS Connection	INTERNET	2018
TCP Socket	INTERNET	2006
HTTP Connections	INTERNET	1987
WebView JavaScript Interface	INTERNET	1738
URL Connection to file/http/ftp	INTERNET	1552
TCP Server Socket	INTERNET	1353
UDP Datagram Socket	INTERNET	1197
UDP Datagram Packet	INTERNET	887
File I/O Operations	FILE OPERATION	2151
DATABASE	Query Database	2058
GPS Location	DEVICE DETAILS	1945
Device Location	DEVICE DETAILS	759
SIM Operator Name	DEVICE DETAILS	1417
SIM Provider Details	DEVICE DETAILS	1401
Device ID, IMEI	DEVICE DETAILS	1258
Device Information	DEVICE DETAILS	648
Subscriber ID	DEVICE DETAILS	484
SIM Serial Number	DEVICE DETAILS	339
Software Version, IMEI/SV	DEVICE DETAILS	74
Send SMS	CALL/SMS	115

Table 6.2: API Statistics

Almost all the applications perform a network request or access to a file, and a lot of them use a method to request the GPS location or include a WebView. More than half of the apps include code to read the Device ID or the IMEI, and only a few of them can send SMS.

In Table 6.3 we report details of the statistics for the permission analysis, which is available for 2356 app versions.

PERMISSION NAME	TYPE	APPS
android.permission.INTERNET	declared	2349
android.permission.ACCESS_NETWORK_STATE	declared	2346
android.permission.WAKE_LOCK	declared	2195
com.google.android.c2dm.permission.RECEIVE	declared	1985
android.permission.WRITE_EXTERNAL_STORAGE	declared	1853
android.permission.ACCESS_NETWORK_STATE	requiredButNotUsed	2345
com.google.android.c2dm.permission.RECEIVE	requiredButNotUsed	1985
android.permission.WRITE_EXTERNAL_STORAGE	requiredButNotUsed	1853
...BIND_GET_INSTALL_REFERRER_SERVICE	requiredButNotUsed	1705
com.android.vending.BILLING	requiredButNotUsed	1632
android.permission.INTERNET	requiredAndUsed	2347
android.permission.WAKE_LOCK	requiredAndUsed	2195
android.permission.ACCESS_WIFI_STATE	requiredAndUsed	1583
android.permission.VIBRATE	requiredAndUsed	1125
android.permission.ACCESS_FINE_LOCATION	requiredAndUsed	765
android.permission.READ_PROFILE	notRequiredButUsed	2214
android.permission.MANAGE_ACCOUNTS	notRequiredButUsed	1597
android.permission.WRITE_SETTINGS	notRequiredButUsed	1502
android.permission.ACCESS_COARSE_LOCATION	notRequiredButUsed	1461
android.permission.GET_TASKS	notRequiredButUsed	1377

Table 6.3: Permission Statistics

INTERNET and *ACCESS_NETWORK_STATE* are the most declared permissions (STAT1) and are used to open a connection and access information about the network. The permission *WAKE_LOCK* is used to keep the screen turned on, *RECEIVE* to receive push notifications, and *WRITE_EXTERNAL_STORAGE* to write to external memory.

The notification permission was used by Google Cloud Messaging, which is now deprecated: in the Google Developers migration guide to Firebase⁴, one of the step explains to remove the permission since obsolete. Requesting unnecessary permissions should be avoided, but besides this permission also the second and the last one are in the list of the most required permissions that are never used with *BIND_GET_INSTALL_REFERRER_SERVICE* and *BILLING*, used to retrieve information from the store before the app installation and managing billing transactions inside the application using Google Play.

⁴<https://developers.google.com/cloud-messaging/android/android-migrate-fcm>

The *INTERNET* and *WAKE_LOCK* permissions are also the most used, followed by *ACCESS_WIFI_STATE*, *VIBRATE*, and *ACCESS_COARSE_LOCATION*. The last one allows the app to access the location using network information from cell towers and Wi-Fi, it is considered a dangerous permission (like the external memory access) and therefore from Android 6.0 the app asks for it during its execution. It is one of those most used but not requested with *READ_PROFILE* to access user profile information, *MANAGE_ACCOUNTS* to access user's online accounts, *WRITE_SETTINGS* to read or write the system settings, and *GET_TASKS*, which is deprecated.

The last group is probably related to libraries code which is not executed since an app is not allowed to access resources without permission, but app collusion techniques [ARCR17] could allow performing actions that a single application is not able to perform. Studying joint malicious action is an interesting threat, however, we were not able to save information about app components due to an implementation bug: we solved it, but not in time to collect useful information for many applications and therefore fail the SECM1 goal.

The most frequent vulnerability (STAT4) reported is the access to the external storage, indeed it is one of the most used permissions above: developers should not save sensitive information to external files and as discussed in Section 6.2.1, any app with the permission to read external memory could potentially access user's personal data like pictures. In the previous section, we also described problems related to the request of the Device ID and the presence of insecure connections, which are also frequent: the first of these vulnerabilities and the previous one are classified as a low-level problem, but the URLs in HTTP has instead high priority since interceptable and alterable.

The other problems are related to the presence of a web view, which allow file access and enable javascript execution, classified respectively as a low and medium level vulnerability. These problems could allow to access local resources or perform malicious code injection through cross-site scripting attacks, but could be simply disabled using the methods in *WebSettings* class.

It follows another web view vulnerability, related to the injection of Java objects into the web page: this could allow calling the methods from JavaScript, but in some cases, it can be used to control the application.

In particular, it is a security risk before Android 4.2⁵: even if Jelly Bean or previous versions are used only by a small percentage of the users (red in Figure 6.4), vulnerable apps should not target the devices. Of these 1700 entries, we can filter those which are dangerous using the query in the Code-Snippet 7 and get more than 1000 results.

⁵[https://developer.android.com/reference/android/webkit/WebView#addJavascriptInterface\(java.lang.Object,%20java.lang.String\)](https://developer.android.com/reference/android/webkit/WebView#addJavascriptInterface(java.lang.Object,%20java.lang.String))

```
vulnerabilities.vulnerability_id = WEBVIEW_RCE; sdk.min <= 17
```

Code-Snippet 7: Query to find vulnerable webview apps due to the target device

The next issues reported are: exported components available to other applications, HTTPS connections performed without the validation of the SSL Certificate, services started using an implicit intent that can be handle by any application and the allowed ADB backup for the app (default behavior). The last one is a problem because the user could perform the backup to external storage without specifying a password, therefore if the application saves sensitive information, credentials or an access token, it should be explicitly forbidden in the manifest. Instead, for the apps affected by the previous report, an exception is thrown from Android 5.0: before this version, the user can not see which service starts and more than 900 apps still target these devices (orange and red in Figure 6.4).

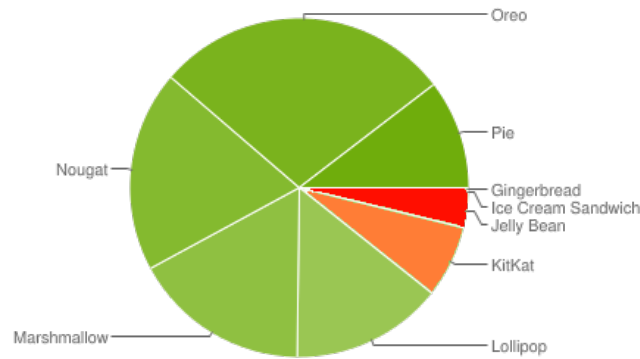


Figure 6.4: Android versions distribution

The most used libraries (STAT3) are *Google Mobile Services*, which contains Google advertising services like DoubleClick, Google Mobile Ads, AdMob or Rewarded Video Ads and the *Android support library version 4*, which is no longer maintained and is recommended to migrate to AndroidX Jetpack. Following the *Google gson* library to convert Java Objects into JSON, the *Google Play Library* to access information from the official market, the development platform *Firebase*, which also provides analytics tools and the *Facebook* library. Next there are the *Zebra Crossing* barcode image processing library, the *OKHttp3.0* client, the no longer maintained *Android support library version 7* and the *Bolts* collection of low-level libraries.

In particular *Google Firebase Analytics*, *Google Ads* and *Google DoubleClick* are reported as the most used advertising libraries with *Facebook Login*, *Google CrashLytics*, *Facebook Share*, *Facebook Analytics*, *Facebook Ads*, *Moat* and *Unity3d Ads*.

In Figure 6.5 we build a graph that connects applications to the tracking libraries, the dimension of a node and its color depends on the in-degree. Performing the metrics as an undirected graph, its diameter is equal to 7 and the average path length to 2.3: since in the path each application is divided by a library, an average distance of two means that a random pair of apps probably has a tracker in common.



Figure 6.5: Advertising libraries graph

The 10 most contacted domains (STAT5) are also related to advertising: *graph.facebook.com*, *settings.crashlytics.com*, *firebase-settings.crashlytics.com*, *analytics.query.yahoo.com*, *firebase-remoteconfig.googleapis.com*, *e.crashlytics.com*, *data.flurry.com*, *s3.amazonaws.com*, *events.appsflyer.com*, *rt.applovin.com* and *ms.applovin.com*.

The dynamic analysis is executed only on a subset of applications and therefore the presented domains are less representative than the previous metrics: the same consideration is valid for the external Memory access (STAT7), reported only for few apps in the *Download*, *Music*, *Movies*, *Ringtones* folder and the file related to advertising configuration */storage/emulated/0/.UTSystemConfig/Global/Alvin2.xml*.

This behavior could represent a security risk because applications with different permissions can potentially exchange sensitive information and for this reason, the statistics on the leaks (STAT8) have a purpose also if concern only a few apps: two domains are involved in a leak from different applications (STAT6).

6.4 Additional Studies

In this section, we describe how Appregator can be used also to retrieve information not directly shown in the platform. The most interesting cases concern the study of differences between stores or in an application after an update. Finally, we try to find if there are permissions for which vulnerabilities are more common: this is useful because permissions are granted by the user and identify which are the most dangerous can help him assess the risk.

There are no big differences in the statistics when applied to the Playstore or to Aptoide: the order of the elements could change, but each one is present without a great alteration of its position. Also when applied on an additional filter, like a specific permission or the use of a certain API, metrics are really similar if there are enough results. We can look for differences in single applications: an approach to decide which apps starting to discuss could be to write a script to retrieve those with a different number of fields for each module.

Implementing this script, we should take note that it is possible that a module fails the analysis, and therefore ignore those not present in both versions. Another option could be to save JSON objects as strings and perform a similarity measure, but obtain this information with long values will take too much time and it is probably not possible on a large set. A simpler method is to order the applications by differences in size between markets: Google Chrome app in the Aptoide store has the half size compared to the Play version and Adobe Scan PDF takes up 63 MB when downloaded from Aptoide, but the APK file on the official market is only 10 MB. Nevertheless, all the metrics are equal for these two examples and it is probably just a different packaging strategy, which could be related to an easier way to refer to different platforms on the official store.

Despite these two examples, dimensions are really different only for few apps and performing the comparison following this order, we have not noticed big differences even for single applications and therefore no interesting change between apps on the two stores are provided. Another issue concerns applications with a lite version, like Facebook: in the smaller alternative, it does not contain the library *RoboGuice* (a dependency injection framework), and the related chat app has less advertising libraries. Above, we discuss the scenarios in the updates of some of the most famous apps: we group them counting the number of versions to retrieve those which perform updates most frequently.

Social networks and streaming apps often release updates, since we analyzed many different versions of the Playstore for Instagram, TikTok, Pinterest, Twitter, Youtube, Spotify, and Netflix. Between the first and the last analyzed version of Instagram, the Firebase library has been removed or no more recognized by the tool together with the permission *com.instagram.android.permission.C2D_MESSAGE*, that is actually related to the Google Cloud Messaging for Android.

Also in TikTok a lot of libraries have been removed: *EventBus*, for communication between activities, threads or services, *Apache Common*, which provides reusable Java components, the XML/JSON parser *Fasterxml* and the *Google API Client Library*. The advertising libraries *Facebook Analytics* and *AccountKit* are no more reported for Pinterest application after an update.

On the contrary, in Spotify the permission *com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE* has been added to retrieve information from the Play Store and an API related to UDP Datagram Socket that was not present previously is used.

For the other mentioned apps, no difference in the measures are found, but just some changes in the position of the indicated vulnerabilities without any additional reports. There are also other apps like Duolingo and TrueCaller that released many updates during our analysis, and for the first one, the above considerations are valid. The other instead removed the not required and used permission *android.permission.CHANGE_WIFI_STATE* in the last version, probably related to the deletion of the advertising libraries *Moat* and *Inmobi*, the percentage of obfuscation of the code also changes from 52.35% to 55.65%.

If we select a specific vulnerability, like the request of the Device ID, the related permission is more common in the filtered results and therefore appears in the statistics for that subset. Since permissions allow the user to decide to which resources grant access to the app, we would like to find which ones are related to a vulnerability. We select the permissions from the 100 analyzed applications of the play store with the highest number of dangerous vulnerabilities, obtained using the limit feature of the API and sorting the results with *sort=-nvuln_risk.high,-nvuln_risk.medium,-nvuln_risk.tot*. Then, we count for each group the most frequent permissions to compare with the general statistics and select those which rise significantly in position.

The permission to read external memory has a higher ranking both for those declared in the manifest and for those not used in the code, in this category also the *com.android.launcher.permission.INSTALL_SHORTCUT* is present, which allows to automatically insert a link to the launcher. The first one is related to storage access vulnerabilities, but we do not find a direct explanation about the other one and it could just be due to the choice to select only 100 results, made to have a big number of vulnerabilities for each app and keep only the most dangerous. However, it is in the *SYSTEM_TOOLS* group and since in Android permissions are granted to the whole group, it could be also related to another one. Also the permissions to access user accounts and change the WiFi state rise in position, which allows to login to an online service and turn on and off the network.

Chapter 7

Conclusion

7.1 Discussion

In one month, Appregator collected information from almost 200.000 applications and performed the security analysis on the 2.000 most installed applications in the Play Store, using only a single machine. It saved more than 3.500 database entries and APK files, including updates and the Aptoide app versions.

Figure 7.1 shows the crawler statistics: it collected almost 100.000 applications after one week, but it needs more than a month to duplicate the result. The labels indicate each step of the collection cycle, the last one was stopped before the conclusion and, after that, also the list of failed applications should be considered. In the first run, more than 10.000 results for the top app categories are returned in about 8 hours, and after five days almost 75.000 new apps are obtained using the most common words search feature.

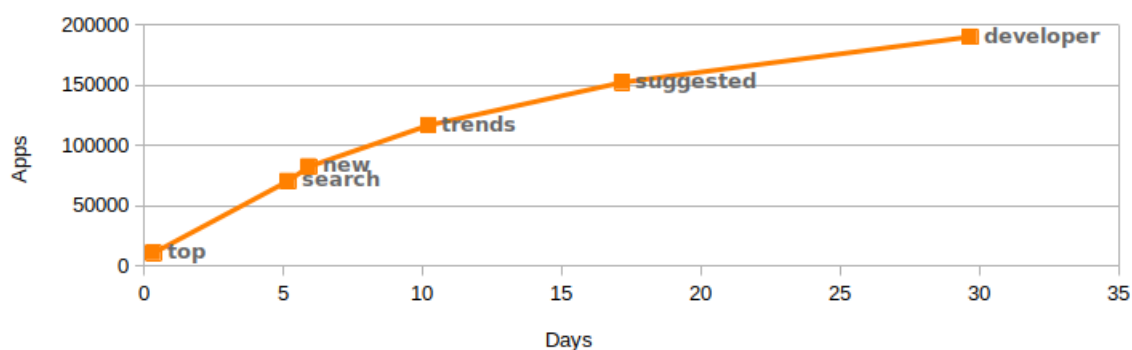


Figure 7.1: Applications collection cycle

Next operations are executed until the container is stopped, we do not know how much time a cycle takes to end, but the first execution should be slower because also the results of the previous step are used to look for similar applications or those distributed by the same developer. The new apps categories and search trends features have analog performances of the first operations, while the others take more time but depend on the output of the previous executions.

The goal of the thesis was to build a platform able to manage large scale study, and therefore we used technologies optimized for big data and we designed the platform with a distributed architecture. Other works (Section 2.3) collected a lot of more data, but reached one million results only after a couple of years [PLN⁺14] or three million in several months using seven nodes and starting from other datasets [ABKLT16]. While finding information about new apps from the store becomes more difficult over time since repetitions often occur, the number of results produced in the analysis step is constant and depends on the performance of the computer (more than 100 new analyses per day in our case).

After comparing the library statistics with the AppBrain list of top development and advertising tools, we noticed that we obtained equivalent results also concerning the order of the elements, especially in the first positions. Consequently, the analyzed apps are relevant for this metric and therefore also the other measures could be valid for a larger set of applications.

In any case, our objective was to consider the most used applications: the produced results show how the static analysis is useful for computing general statistics, while the dynamic tests are essentials to find security and privacy vulnerabilities. Indeed, using the first technique we showed how to retrieve a list of potentially vulnerable applications related to a specific issue. The second technique allowed to retrieve details about insecure connections, privacy leaks and hard-coded keys for some specific apps.

Moreover, we discussed how the platform can be used to find differences between app versions after an update or when it is downloaded from different stores. In particular, we did not find any increase in the number of vulnerabilities for the applications retrieved from Aptoide, and therefore we consider the market secure for downloading the top apps. Finally, we proposed a method to find permissions most at risk and listed the obtained results.

7.2 Future Work

The main objective is to perform more analyses to produce new security reports and keep collecting information about new apps from the Play Store; another improvement could be to retrieve applications also from different stores, to obtain a score for the most reliable.

However, the goal of the project remains to download files only from important markets like APK Mirror, Amazon, Huawei or Galaxy Store and F-droid. The last one collects open-source versions of apps: studying differences in the number of vulnerabilities and privacy leaks in a store that does not allow advertising could be an interesting research question.

Also adding new modules to the worker is an interesting challenge: despite we parse the analysis results to retrieve only the most interesting fields, and adding code to the platform to keep it updated with new Approver versions is the best method, we provide future support implementing a worker feature not discussed yet. In particular, Appregator contacts all the available modules, but saves the results only for the known ones: if in the response of a new module is specified a flag *Appregator* with its name, such results are likewise saved.

The main limitation of the analysis is to find interesting results during the dynamic tests: in fact, it is difficult because often the user should perform a login to access all the features. The installation of new modules able to overcome this problem, defining procedures that use machine learning to identify the fields to fill, should allow producing much more results.

At last, the obvious objective is to use the platform to find new issues and share it with researchers to find more complex correlations between data to predict security and privacy vulnerabilities based on application features.

Bibliography

- [ABKLT16] Kevin Allix, Tegawendé Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. pages 468–471, 05 2016. doi:10.1145/2901739.2903508.
- [AEM⁺19] Amr Amin, Amgad Eldessouki, Menna Magdy, Nouran Abdeen, Hanan Hindy, and Islam Hegazy. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10, 10 2019. doi:10.3390/info10100326.
- [ARCR17] F. I. Abro, M. Rajarajan, T. Chen, and Y. Rahulamathavan. Android application collusion demystified. *Communications in Computer and Information Science*, 759:176–187, August 2017. doi:10.1007/978-3-319-65548-2_14.
- [BBD16] Michael Backes, Sven Bugiel, and Erik Derr. pages 356–367, 10 2016. doi:10.1145/2976749.2978333.
- [bIT17] Ponemon Institute LLC Sponsored by IBM and Arxan Technologies. 2017 study on mobile and internet of things application security. Technical report, 2017. URL: <https://www.ponemon.org/local/upload/file/Arxan%20Report%20Final%205.pdf>.
- [Cha19] Raj Chandel. Docker privilege escalation. Technical report, 2019. URL: <https://www.hackingarticles.in/docker-privilege-escalation/>.
- [Coo19] Fiddly Cookie. How to pull-off a nosql injection attack. Technical report, 2019. URL: <https://medium.com/@shukla.iitm/nosql-injection-8732c2140576>.
- [DBF⁺17] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. pages 2187–2200, 10 2017. doi:10.1145/3133956.3134059.

- [DD16] B Sharma D Das. Static and dynamic analysis of android applications for detection of android packet misbehaviour. *International Journal of Computer Science and Information Security (IJCSIS)*, 14, 11 2016.
- [Der18] Erik Derr. The impact of third-party code on android app security. In *Enigma 2018 (Enigma 2018)*, Santa Clara, CA, January 2018. USENIX Association. URL: <https://www.usenix.org/node/208134>.
- [Ege19] Serge Egelman. Ad ids behaving badly. Technical report, 2019. URL: <https://blog.appcensus.io/2019/02/14/ad-ids-behaving-badly>.
- [Far18] Umer Farooq. Android operating system architecture. 07 2018. doi:10.13140/RG.2.2.20829.72169.
- [GM18] Franz-Xaver Geiger and Ivano Malavolta. Datasets of android applications: a literature review. 09 2018.
- [HRM⁺16] Johannes Hoffmann, Teemu Ryttilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. pages 139–141, 03 2016. doi:10.1145/2857705.2857737.
- [KJ18] Keyur Kulkarni and Ahmad Javaid. Open source android vulnerability detection tools: A survey. 07 2018.
- [LBP⁺17] Li Li, Tegawendé Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Ocateau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 04 2017. doi:10.1016/j.infsof.2017.04.001.
- [LGH⁺17] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. 09 2017.
- [LWW⁺18] M. Li, P. Wang, W. Wang, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo, and W. Zou. Large-scale third-party library detection in android markets. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. doi:10.1109/TSE.2018.2872958.
- [MXS⁺17] Guozhu Meng, Yinxing Xue, Jing Siow, Ting Su, Annamalai Narayanan, and Yang Liu. Androvault: Constructing knowledge graph from millions of android apps for automated computing. 11 2017.

- [PLN⁺14] Christian Platzer, Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, and Victor van der Veen. Andrubis - 1,000,000 apps later: A view on current android malware behaviors. 09 2014. doi:10.1109/BADGERS.2014.7.
- [QWR18] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 176–186, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213873.
- [Wei18] Daniel Weibel. Docker in docker? Technical report, 2018. URL: <https://itnext.io/docker-in-docker-521958d34efd>.
- [Yat19] Ravali Yatham. Node.js memory management in container environments. Technical report, 2019. URL: <https://developer.ibm.com/articles/nodejs-memory-management-in-container-environments/>.