



**HAL**  
open science

## APPregator: A Large-Scale Platform for Mobile Security Analysis

Luca Verderame, Davide Caputo, Andrea Romdhana, Alessio Merlo

► **To cite this version:**

Luca Verderame, Davide Caputo, Andrea Romdhana, Alessio Merlo. APPregator: A Large-Scale Platform for Mobile Security Analysis. 32th IFIP International Conference on Testing Software and Systems (ICTSS), Dec 2020, Naples, Italy. pp.73-88, 10.1007/978-3-030-64881-7\_5. hal-03239815

**HAL Id: hal-03239815**

**<https://inria.hal.science/hal-03239815v1>**

Submitted on 27 May 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# APPregator: a Large-scale Platform for Mobile Security Analysis \*

Luca Verderame<sup>1</sup>[0000-0001-7155-7429], Davide Caputo<sup>1</sup>[0000-0002-5408-4735],  
Andrea Romdhana<sup>1,2</sup>[0000-0003-4651-8713], and Alessio  
Merlo<sup>1</sup>[0000-0002-2272-2376]

<sup>1</sup> DIBRIS, University of Genoa, Genoa, Italy  
{name.surname}@dibris.unige.it

<sup>2</sup> Security & Trust Unit, FBK-ICT, Trento, Italy

**Abstract.** The Google Play Store currently includes up to 2.8M apps. Nonetheless, it is rather straightforward for a user to quickly retrieve the app that matches her tastes, as Google provides a reliable search engine. However, it is likewise almost impossible to select apps according to a security footprint (e.g., all apps that enforce SSL pinning). To overcome this limitation, this paper presents APPregator, a platform which allows security analysts to i) download apps from multiple app stores, ii) perform automated security analysis (both static and dynamic), and iii) aggregate the results according to user-defined security constraints (e.g., vulnerability patterns). The empirical assessment of APPregator on a set of 200.000 apps taken from the Google Play Store and Aptoide suggests that the current implementation grants a good level of performance and reliability. APPregator will be made freely available to the research community by the end of 2020.

**Keywords:** App Analysis · Static and Dynamic Analysis · Security & Privacy

## 1 Introduction

Risk assessment is crucial for IT systems, and it became a key issue for mobile apps, 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 a global view of the security status over a set of arbitrarily selected mobile apps. In particular, some research projects produced datasets that allow performing analysis on the security features of apps, however, these projects are focused mainly on 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 apps that could be exploited both

---

\*This work was partially funded by the Horizon 2020 project “Strategic Programs for Advanced Research and Technology in Europe” (SPARTA).

by malicious apps or external attackers (i.e., confused deputy attacks). Albeit there exist some proposals that allow assessing the number of apps affected by a specific vulnerability, the results and the tools are kept mostly private so that only a few information is available and the tools cannot be used to replicate the analysis in the wild.

To overcome previous limitations, we designed and implemented a large scale platform for mobile security analysis called **APPregator**. The purpose of APPregator is to help researchers and developers to discover new vulnerabilities in Android apps, allowing them to aggregate and analyze a subset of Android apps that match specific security and privacy constraints as well as examining the distribution of security flaws. The proposed platform can automatically download apps from multiple stores and aggregate information about them, perform automated security analysis (both static and dynamic), and moreover, it is able to notify the results of the security analysis to apps' developers in a fully automatic way. Furthermore, our proposal allows researchers to aggregate the results according to defined security constraints and privacy.

To prove the scalability and reliability of our proposal, we collected in one month 200,000 apps information from app stores and we performed a security analysis on 3,500 apps. Furthermore, the 3,500 apps were subjected to statistical analysis with the aim of examining the distribution of security flaws (i.e., insecure connections, privacy leaks, and hard-coded keys).

***Structure of the paper*** The rest of the paper is organized as follows: Section 2 introduces the vulnerability assessment of mobile apps, while Section 3 presents the current state of the art. Section 4 defines the requirements of a large-scale platform, Section 5 presents the design of APPregator. Section 6 discusses the implementation choices for APPregator, while Section 7 shows the capabilities of the platform proposed. Finally, the Section 8 concludes the paper and points out some extensions of this work.

## 2 Vulnerability Assessment of Android Apps

The vulnerability assessment of Android apps is based on two main approaches: static and dynamic analysis. Static analysis techniques enable the evaluation of an app by examining the source code and the package content without executing it. Static analysis examines all possible execution paths and variable values, not just those invoked during execution. Thus, it can reveal errors and vulnerabilities that may not manifest themselves during the code execution, thereby leading to false positives.

Dynamic analysis techniques, instead, explore and evaluate the behavior of the app during its execution. Unfortunately, its effectiveness is influenced by the current limitations imposed by automated testing techniques that fail to analyze the whole app surface, leading to false negatives.

To this aim, the typical assessment workflow combines both the approaches, and it is composed of the following steps:

- *APK Reverse Engineering.* In this step, the Android Package (APK) is unpacked to retrieve manifest, resources, certificate and the app compiled code (dex files). The compiled code can be disassembled or decompiled. In the first case, the dex files are transformed using a disassembler (e.g., apktool [1]) in order to obtain a human-readable representation of the dex format called Smali [14]. In the second case, the dex files are reconverted in the original source code (or a representation of it) using tools like dex2jar [7] or jadx [11].
- *Static Analysis.* The app code is analyzed to evaluate the presence of security misconfigurations, vulnerable APIs, and security patterns. Static analysis of an app package often relies on the model representation of the app logic (e.g., through control flow graphs or call graphs [45,19]). Furthermore, static analysis techniques also evaluate non-code elements, i.e., the manifest file and all the resources files. These files are analyzed to detect misconfiguration or essential information about the app (e.g., exported activities, debug information, hard-coded keys, etc.). The most famous used tools are FlowDroid [21], TaintDroid [26] and AmanDroid [41].
- *Dynamic Analysis.* During the dynamic analysis, the app is installed and executed in a controlled environment (usually a virtual machine) in order to monitor its runtime behavior. During this phase, the behavior observed is the interaction with the file system, with the OS (i.e., all APIs invoked at runtime), and with the network by sniffing and collecting all traffic exchanged by the app during the execution. The app is stimulated using automatic tools that simulates human behavior like Droidbot [33] or Monkey [12].

### 3 Related Work

The typical vulnerability assessment workflow has been adopted by several frameworks and tools to either evaluate the security state of the apps (e.g., SCanDroid [28], CHEX [34], DroidChecker [23], DroidSafe [30], AppAudit [43]) or to monitor apps behavior (e.g, ProfileDroid [42], CopperDroid [39], AppIntent [46], DroidMiner [44], AsDroid [31]). However, all the above solutions focus on the security evaluation of a single app and do not provide mechanisms to i) retrieve sets of apps automatically, ii) evaluate the app’s security posture in the different versions and the various application stores, and iii) prune, query and filter apps according to their security evaluation.

In the last years, both the scientific and industrial communities have proposed datasets that collect security information about apps. In [29] the authors identified that there are five datasets that collect information of more than 1 million of apps: Appannie [3], AppBrain [2], AndroZoo [32], AndroVault [38], and Andrubis [35]. 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 enriched with some metadata and analyzed with several static or dynamic analysis techniques. For instance, AndroZoo collects information from: (1) the Google Play Store,

(2) the manifest file, (3) the *classes.dex* files, (4) the presence of native and encrypted code, and (5) the reports of the antivirus and vulnerabilities detection services. Unfortunately, Meng et al. [35] claim that it is complex to extract valuable information from the AndroZoo dataset and therefore they build a graph of attributes to extract these data. Androvault and Andrubis are two datasets that also perform dynamic analysis. Still, in the former, the authors were unable to execute the dynamic analysis of many apps due to time and technological constraints. The latter, instead, is discontinued and it is no longer updated from 2015. All in all, the primary purpose of those datasets is to analyze and study mobile malware, where most of them are obtained from alternative app stores or virus collections. To the best of our knowledge, there are no solutions able to aggregate apps coming from several stores (including the Google Play Store) and perform both static and dynamic analysis in order to make available to the researchers and developers community the security state of the app and its evolution.

## 4 Requirements for a Large-scale Platform

In this section, we identify the requirements for a large-scale platform with the capability to perform automated security analysis (both static and dynamic) and aggregate the results according to user-defined security constraints. In detail, the aim of our proposal is to design and implement an architecture that is able to:

1. automatically collect a large set of mobile apps from several app stores;
2. perform static and dynamic analysis of those apps following the typical workflow, as described in Section 2;
3. store all the security-relevant information of the analysis to build a security dataset that can be queried and used to infer statistical results;
4. automatically check the release of app updates to create a security history of mobile apps releases, and
5. automatically notify the developers about security issues founded in their apps.

***Apps Collection, Update & Filtering.*** Since the architecture needs to retrieve large sets of apps from the application stores automatically, it must crawl and download app packages from the leading Android app stores, i.e., Google Play and Aptoide [4]), being two of the most widespread Android market places. For each analyzed app, the architecture should also collect a set of app metadata, to enable fine-grained filtering of the dataset. In detail, we are interested in recording: (1) package name, (2) title, (3) icon, (4) category, (5) developer, (6) number of installs, (7) privacy policy, (8) number of ratings, (9) average ranking, (10) existence of advertising, (11) timestamp of the last update, (12) market of origin, (13) app version, (14) file hash, and (15) the analysis date.

**Security Analysis.** The architecture needs to perform a systematic security evaluation of the Android apps, thereby adopting the workflow described in Section 2. To this aim, the platform needs to perform *Reverse Engineering*, *Static Analysis*, and *Dynamic Analysis*. The reverse engineering step is necessary to obtain information about the files that compose the app and the structure of the app itself (i.e., activities, services, providers, and receivers). Regarding the static analysis, we are interested in evaluating the following information:

- the permissions (used and requested),
- the invoked APIs;
- the presence of patterns that could bring to security vulnerabilities. Such a vulnerability analysis should follow security frameworks and standards, e.g., the OWASP Mobile Applications Security Verification Standard (MASVS) [37] and the NIST Mobile Threat Catalogue [36];
- the usage of third-party libraries or frameworks;
- the obfuscation level;
- the presence of malicious code.

Regarding the dynamic analysis techniques, our platform needs to cover all of the main security measurements. Since most apps communicate over the Internet, the security assessments of network traffic and the communication protocols [20] is an essential feature. Among the possible checks, the platform needs to evaluate the absence of unencrypted communication and the presence of miss configuration, such as permissive certificate checks or the lack of security enforcement mechanisms (e.g., SSL Pinning) [27]. Furthermore, the platform should also identify the domain reputation of all the accessed URLs. Such a feature would help to detect privacy and security leaks affecting multiple apps by checking if they interact with the same unsafe endpoint.

The proposed platform also needs to detect all vulnerable third-party components since unpatched and obsolete third-party libraries are the primary source of security vulnerabilities in mobile apps, as described in [22,24,40]. The platform should also report the usage of tracking or advertising libraries to evaluate the presence of potential privacy leaks.

Furthermore, the dynamic analysis performed by the platform should also include additional information such as i) the application access to internal and external memory files, ii) the disclosure of sensitive information in the system and application logs, and iii) the use of hard-coded keys, passwords, or private token. Besides the filtering and sorting features, the system should also provide general security statistics such as the most commonly used libraries and contacted domains or the most requested files and leaks found at the same location reported for more apps.

**Developers Notification.** The platform needs to follow the responsible disclosure guidelines imposed by both application stores and the involved vendors. In detail, we designed our platform to be compliant with the Google’s vulnerability disclosure policy [10]. In order to meet the requirements, after each security

analysis, the platform automatically notifies the app’s developer, waiting for 90 days for the inclusion in the dataset. The developer will be able to evaluate the security analysis and apply the required fixes to their apps within that time.

**Platform Requirements.** To formalize the platform requirements, we organize its features in functional requirements (FUNR), security measures (SECM), and statistical measures (STAT). The first group indicates the primary functionalities of our platform; the second one includes all the security measurements that need to be extracted by the mobile apps, while the last group collects the global metrics that can be computed by the platform. Table 1 presents the full list of requirements and their corresponding IDs.

ID	Description
FUNR1	Filter apps using the package name or a keyword in the title.
FUNR2	Filter apps from a specific developer or category.
FUNR3	Filter apps by the number of installations or their score.
FUNR4	Filter apps that do not include a privacy policy on the Google Play Store page.
FUNR5	Filter apps that requires a specific permission.
FUNR6	Filter apps that use a certain category of APIs.
FUNR7	Filter apps that include a specific library.
FUNR8	Filter apps that use a low obfuscation level.
FUNR9	Filter apps that interact with a specific internet domain.
FUNR10	Filter apps that use insecure connections.
FUNR11	Filter apps that access to a specific file path.
FUNR12	Filter apps for which a leak is reported or an hard-coded key is found.
FUNR13	Notify the app developers of security issues found in their app.
SECM1	The list of components included in an app
SECM2	The list of permissions required by an app.
SECM3	The list of APIs used by an app.
SECM4	The list of libraries included in an app.
SECM5	The request to read the Device ID.
SECM6	The list of vulnerabilities reported for an app.
SECM7	The obfuscation level of an app.
SECM8	The probability for an application to be a malware.
SECM9	The list of hosts contacted by an app with network security settings.
SECM10	The list of files in the external memory used by an app.
SECM11	The list of leaks found for an app.
SECM12	The list of hard-coded keys reported for an app.
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 1: Platform Requirements

## 5 APPregator: a large-scale platform for mobile security analysis

In order to meet all requirements described in the previous section, we propose a modular architecture called APPregator. APPregator is composed of several

modules, namely **Database**, **Crawler**, **Worker**, and **Server**. These modules allow to retrieve app information, to analyze its security, and to manage the obtained reports and statistics.

The **Database** stores all apps information, while the **Crawler** module identifies and collects the mobile apps. The Crawler also keeps track of the various versions of the apps. The **Worker** downloads the apps and performs the security analysis. A set of rest API is exposed by the **Server** to interact with the platform; moreover, the **Server** features a responsive GUI to extract and filter the data. Figure 1 depicts the modules of APPregator.

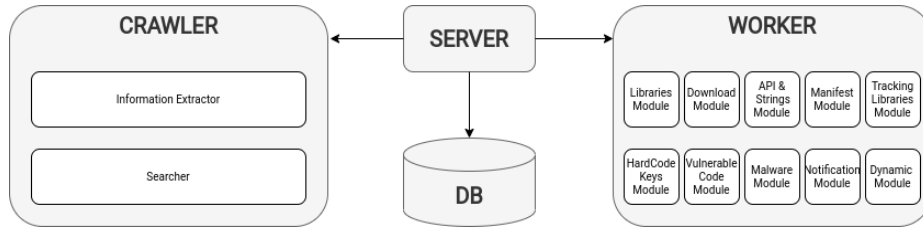


Fig. 1: APPregator architecture

*Crawler.* The Crawler uses the sub-module *Searcher* to collect all free apps that appear in Store categories. Once started, the *Searcher* looks for new apps in the categories TOP FREE and GROSSING. The keywords used come from the most common terms provided by the sub-module *Information Extractor*. *Information Extractor* relies on the Google Trends [9] service to function properly. At last, the *Searcher* starts examining TRENDING and NEW FREE categories searching for apps belonging to the same developer or similar to those already chosen during the current step. The similarities between apps are suggested by the store in use.

*Worker.* The Worker is structured as a micro-service, where each security measure is conducted by a specific sub-module (Figure 1). The *Manifest* module retrieves information about app components and permissions (SECM1, SECM2). The *API & Strings* module is able to retrieve all APIs and string used by the app dividing them into categories (SECM3). The list of libraries included in the app are retrieved by two sub-modules: the *Libraries* and the *Tracking Libraries* (SECM4). The *Vulnerable Code* sub-module reports all vulnerabilities found in the app (SECM6), together with the obfuscation level of the app code (SECM7). The virus scan is performed by the *Malware* sub-module (SECM8). The *Dynamic* sub-module performs the dynamic analysis and stores the list of: (1) the host contacted by the app (SECM9), (2) the files used by the app (SECM10) and (3) the information leaked (SECM11). The Device ID access (SECM5) can be retrieved combining the information obtained from Vulnerable Code module and the Dynamic module, while the *HardCoded Keys* module detects the usage of hard-coded keys (SECM12).



The *Download* service downloads apps and checks if the latest version available in Google Play Store matches the one in Aptoide. The comparison is done through a specific API [5]. The version of an app is denoted by a name (`verName`) and a code (`verCode`). The Worker compares the SHA-2 of the downloaded APKs from the Play Store with those that have the same `verName` coming from unofficial markets. If the hashes are different, the Worker performs a new security analysis. If the apps have the same `verName` and the same `verCode` but a different hash, the Worker executes the malware analysis, to detect whether those apps are embedding malicious code.

Once the analysis is completed, the Worker uses the **Notification** sub-module to send the analysis result to the app developer. At last, the Worker publishes the results on the platform, after the 90-day period imposed by the Google policy has expired (**FUNR13**).

*Database.* The Database stores the information collected during the analysis and provides the server with data which allows the user to execute the queries. The stored data are organized into two tables. One contains information available in app stores about apps (i.e., Play Store, Aptoide), in which each app is identified by the `appId` (package name). The other one contains the results of the security analysis, which has a unique `verCode` (version) for each app associated to the source `store`. The schema of the **Database** is described in Figure 2.

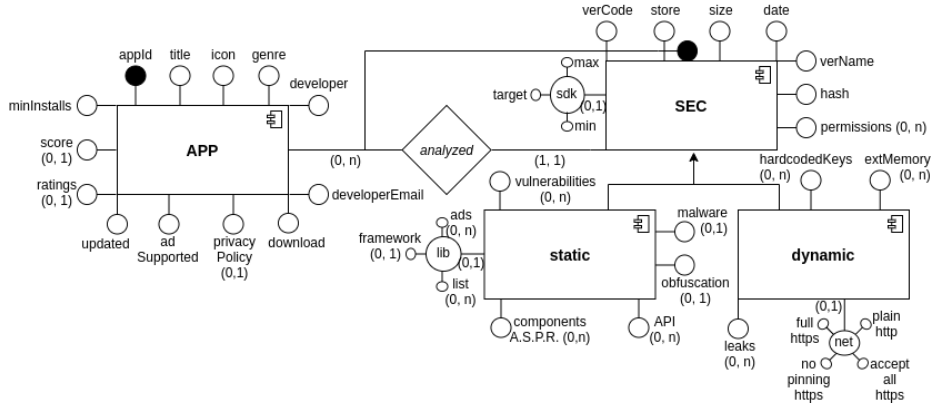


Fig. 2: Conceptual Schema

*Server.* The Server exposes an endpoint of the REST API and a web interface that facilitates searching, sorting, and selecting the results of the application database.

The **Server** also adopts operators typically implemented in the query language, such as count, limit and offset, in order to improve user experience and prevent

The screenshot shows the APPregator web interface. At the top, there's a navigation bar with 'APPregator' on the left and 'List' and 'SEC' on the right. Below this is a search bar with 'Category = Social' and a green 'Search' button. The results show 3647 results. A table lists the following applications:

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	

Fig. 3: Example of APPregator Web Interface

the retrieving of large sets. If the client filters the data, an estimate of the number of query results is returned. Several navigation buttons allow to navigate through the results (Figure 3). In addition, the server offers the possibility to download all the filtered results which could also represent the entire dataset: the server provides a stream from the database to the client to avoid storing the object in the main memory.

## 6 Implementation

The APPregator platform has been designed by leveraging state-of-the-art technologies and tools. The following section explains the implementation of each module.

**Worker.** The **Worker** consists of a set of micro-services developed in Java and Python languages. The security analysis is performed through Approver, a SaaS developed by Talos [15]. Approver consists of:

- The *Decompiler*. It extracts the code and the resources from the app. The code is disassembled from *.dex* to *Smali* format. The resources (e.g., the Android manifest and the accessible strings) are stored on the disk.
- *Manifest Parser*. It analyzes the Android manifest and produces a JSON file containing permissions, activities, services, version and name of the app.
- *String & API Analysis*. This module analyzes the code searching for strings. Moreover it categorizes the Android API calls contained in the app.
- *Permission Checker*. It produces a list of permissions, taking into account both those used in the code and those declared in the manifest.
- *Library Detector*. This module searches for known third-party libraries inside the app.

- *Vulnerability Checker*. It identifies code patterns that could be considered as security risks, classifying their severity according to the OWASP Mobile Top 10 [13].
- *Dynamic Analysis*. This module stimulates the app by simulating human behavior. The aim is to evaluate as many app features as possible to monitor and analyze all API invocations, file system interactions, and network traffic generated. This information is required to identify security vulnerabilities or privacy leaks (e.g., SSL pinning misconfiguration, phone number leaks, etc.). Dynamic Analysis is the most expensive module in terms of resources and time.

The **Worker** uses a heuristic based on the results of the *API & String* module in order to perform the *Dynamic Analysis* module only when it is necessary. In detail, the Worker executes it only if the app uses at least one APIs in at least three of the following categories: INTERNET, FILE, CRYPTO, and PHONE. The Worker is composed also by another module named *HardCoded Keys*. It verifies the presence of hard-coded keys and monitors the functions categorized as CRYPTO. In details, this module compares the strings extracted from the app with the arguments passed to the caller. We decided to ignore arguments with fewer than four characters in order to reduce false positives, because they were too short to be considered as tokens. We also saved the most frequent results for creating lists of exclusion and false-positive.

**Databases.** The Database Management System of APPregator relies on MongoDB. The choice of a document-based solution (i.e., MongoDB) allow clients to query nested values efficiently. The database empowers multi-key index to seamlessly support the addition of new analysis modules or a new crawler. This solution prevents already existing data from being invalidated.

**Crawler & Server.** APPregator exploits the npm library `google-play-scraper`<sup>3</sup>, and the `query-to-mongo` library<sup>4</sup> to extract app data from the Google Play Store. In this way, the users can filter the apps in the **Database** that match specific security-constraint. As a result of the libraries' choice, we used Node.JS and the framework Express for the HTTP server. We used Docker [8] in order to provide portability and fast deployment of the platform. Docker allows to stop (and eventually restart) the containers when a failure occurs or when a task is completed, thereby reducing memory usage.

## 7 APPregator Query Capabilities

We tested APPregator, on real-world apps extracted from the app stores between November 2019 and February 2020. The analysis has been carried out on a single machine with a quadcore 3.40 GHz processor, an SSD with 256 GB of storage,

<sup>3</sup><https://github.com/facundoolano/google-play-scraper>

<sup>4</sup><https://www.npmjs.com/package/query-to-mongo>

and 24GB of RAM. Using this setup, we were able to collect in one month 200.000 apps information from the market and analyzed 3500 different app versions from the app stores. All queries described in this section were executed in a time-span between 200ms and 800ms. The analysis was carried out inspecting two different aspects of mobile apps: Privacy and Security.

### 7.1 Privacy

Using the Web Server 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 1. By using Code-Snippet 1 query on

---

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

---

Code-Snippet 1: Query to find apps with advertising and without a privacy policy

the entire dataset, APPregator reports that almost 13.000 apps 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 out by the number of installations by clicking on the table column name (FUNR3): four apps have at least 100 or 50 million users (Figure 4), 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. Using a search bar the operator can look for apps involved





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	

Fig. 4: Top 4 most installed apps without a privacy policy on the Google Play Store

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 the *Android best practices for unique identifiers* [6] recommends developers to avoid using hardware identifiers like the device ID and use GUIDs to identify app instances uniquely, we inspected the collected dataset. Figure 5 presents the results obtained by the query in the Code-Snippet 2: 9 Google Play Store apps send the Device ID to a server.

---

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

---

Code-Snippet 2: 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* [25] 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 pointed out in 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.

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

Fig. 5: Play Store apps that send the Device ID to a server

## 7.2 Security

Further we tested some of the capabilities offered by APPregator to evaluate the security of the app.

**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 7 results. Among the vulnerable apps, six of them come from the Play Store and have between 10 and 100 million installations. One of them 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 app 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 http URLs. The connection is related to the usage of the *Tencent Login* library [16], indeed APPregator also collects the reference to which part of code is involved to facilitate researchers to discover security issues.

**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, APPregator 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 (see, e.g., [18]) 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 5 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. [35] that allows identifying repackaging of an app by checking the developer certificate w.r.t. the one in the Play Store, we identify an app 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 hash value of apps on Virus Total, we have retrieved an app analysis that reports an `Android.PUA.DebugKey` issue. This issue indicates that the app has a debug certificate: the Play Store would not have allowed the app upload, which is instead possible on Aptoide. However, the differences between the app within Play Store and within Aptoide are all related to third parties libraries: it looks like a patched version of the app with ads removed. The other 4 results are false positives.

**Hard-coded Keys.** APPregator reports 8 apps that contain hard-coded keys (FUNR12): as explained in Section 5, we inserted 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 within 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.

## 8 Conclusion

In one month, APPregator was able to collect information from almost 200.000 apps and performed the security analysis on the 2.000 most installed apps 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. The aim of this work was to build a platform able to manage large scale studies, and therefore we used technologies optimized for big data and we designed the platform with a distributed architecture. Other works (Section 3) collected a lot of more data, but reached one million results only after a couple of years [38] or three million in several months using seven nodes and starting from other datasets [17]. 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 [2] 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 apps. In any case, our objective was to build an architecture able to download, analyze, aggregate a large number of apps in a totally automatic way. The produced results by our architecture show how the static analysis is useful for computing general statistics, while the dynamic tests are essentials to find security and privacy vulnerabilities.

**Future Work.** First, we aim to make APPregator available to the research community by the end of 2020. APPregator can be extended with new modules in order to carry out new types of analysis. Furthermore, APPregator can be extended to retrieve apps from other stores (e.g., APK Mirror, Amazon, Galaxy Store, or F-Droid) and to support iOS apps. The main limitation of our proposal is related to the results obtained during the dynamic analysis, as a full exploration of the app requires to bypass login activities with valid user’s credentials. Currently, APPregator is unable to bypass this step. However, we argue that new modules, leveraging machine learning techniques to identify the fields to fill, can overcome this limitation.

## References

1. Apktool. <https://ibotpeaches.github.io/Apktool/>, accessed: 2020-05-27
2. App brain. <https://www.appbrain.com/>, accessed: 2020-05-27
3. Appannie. <https://www.appannie.com/>, accessed: 2020-05-27
4. Aptoide. <https://en.aptoide.com/>, accessed: 2020-05-27
5. Aptoide api. <https://co.aptoide.com/webservices/docs/7/apps/search>, accessed: 2020-05-27
6. Best practices for unique identifiers. <https://developer.android.com/training/articles/user-data-ids>, accessed: 2020-05-27
7. Dex2jar. <https://github.com/pxb1988/dex2jar>, accessed: 2020-05-27
8. Docker. <https://www.docker.com/>, accessed: 2020-05-27
9. Google trends. <https://trends.google.it/trends>, accessed: 2020-05-27
10. Google’s vulnerability disclosure policy. <https://www.google.com/about/appsecurity/>, accessed: 2020-05-27
11. Jadx. <https://github.com/skylot/jadx>, accessed: 2020-05-27
12. Monkey runner. <https://developer.android.com/studio/test/monkeyrunner/>, accessed: 2020-05-27
13. Owasp mobile top 10. <https://owasp.org/www-project-mobile-top-10/>, accessed: 2020-05-27
14. Smali. <https://github.com/JesusFreke/smali/wiki>, accessed: 2020-05-27
15. Talos sec. <https://talos-sec.com/>, accessed: 2020-05-27
16. Tencent. <https://intl.cloud.tencent.com/>, accessed: 2020-05-27
17. Allix, K., Bissyandé, T., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of Android apps for the research community (2016)
18. Aonzo, S., Georgiu, G.C., Verderame, L., Merlo, A.: Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX* **11**, 100403 (2020). <https://doi.org/https://doi.org/10.1016/j.softx.2020.100403>, <http://www.sciencedirect.com/science/article/pii/S2352711019302791>
19. Armando, A., Costa, G., Merlo, A., Verderame, L.: Enabling byod through secure meta-market. *WiSec ’14*, Association for Computing Machinery, New York, NY, USA (2014)
20. Armando, A., Pellegrino, G., Carbone, R., Merlo, A., Balzarotti, D.: From model-checking to automated testing of security protocols: Bridging the gap. In: Brucker, A.D., Julliand, J. (eds.) *Tests and Proofs*. pp. 3–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
21. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Outeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* (2014)
22. Backes, M., Bugiel, S., Derr, E.: Reliable third-party library detection in android and its security applications (2016)
23. Chan, P.P., Hui, L.C., Yiu, S.M.: Droidchecker: analyzing android applications for capability leak. In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012)
24. Derr, E.: The impact of third-party code on android app security. In: *Enigma 2018* (Enigma 2018). USENIX Association (2018)
25. Egelman, S.: Ad ids behaving badly. *Tech. rep.* (2019)
26. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* (2014)



27. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: An analysis of android ssl (in)security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12, Association for Computing Machinery, New York, NY, USA (2012)
28. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications. Manuscript, Univ. of Maryland, <http://www.cs.umd.edu/avik/projects/scandroidasca> (2009)
29. Geiger, F.X., Malavolta, I.: Datasets of android applications: a literature review (2018)
30. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
31. Huang, J., Zhang, X., Tan, L., Wang, P., Liang, B.: Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In: Proceedings of the 36th International Conference on Software Engineering (2014)
32. Li, L., Gao, J., Hurier, M., Kong, P., Bissyandé, T., Bartel, A., Klein, J., Le Traon, Y.: Androzoo++: Collecting millions of android apps and their metadata for the research community (2017)
33. Li, Y., Yang, Z., Guo, Y., Chen, X.: Droidbot: a lightweight ui-guided test input generator for android. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE (2017)
34. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on Computer and communications security (2012)
35. Meng, G., Xue, Y., Siow, J., Su, T., Narayanan, A., Liu, Y.: Androvault: Constructing knowledge graph from millions of android apps for automated computing (2017)
36. NIST: Mobile Threat Catalogue. <https://pages.nist.gov/mobile-threat-catalogue> (2018), (Accessed on September 2020)
37. OWASP: OWASP Mobile Security Testing Guide. <https://owasp.org/www-project-mobile-security-testing-guide/> (2020), (Accessed on September 2020)
38. Platzner, C., Lindorfer, M., Neuschwandtner, M., Weichselbaum, L., Fratantonio, Y., van der Veen, V.: Andrubis - 1,000,000 apps later: A view on current android malware behaviors (2014)
39. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: Automatic reconstruction of android malware behaviors. In: Ndss (2015)
40. Verderame, L., Caputo, D., Romdhana, A., Merlo, A.: On the (un)reliability of privacy policies in android apps. In: Proc. of the IEEE International Joint Conference on Neural Networks (IJCNN 2020) (2020)
41. Wei, F., Roy, S., Ou, X.: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security (2014)
42. Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Profiledroid: multi-layer profiling of android applications. In: Proceedings of the 18th annual international conference on Mobile computing and networking (2012)
43. Xia, M., Gong, L., Lyu, Y., Qi, Z., Liu, X.: Effective real-time android application auditing. In: 2015 IEEE Symposium on Security and Privacy. IEEE (2015)
44. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: European symposium on research in computer security. Springer (2014)

45. Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W.: Appcontext: Differentiating malicious and benign mobile app behaviors using context. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1. IEEE (2015)
46. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (2013)