university of groningen / faculty of science and engineering / bernoulli institute

Ultimate feature extractor for Android mobile applications

Contextual and Technical Features

Bachelor's Thesis in Computing Science

July 2020

Student: Yona B. Moreda

First supervisor: Fadi Mohsen, PhD

Second assessor: Fatih Turkmen, PhD

Abstract

To combat malware on mobile devices, researchers employ a variety of data gathering and analysis tools to investigate and classify Android device applications based on malicious behavior. As the data is gathered and analyzed, researchers employ a variety of different tools and gather features using many different ways. These different methods of gathering the features produces some inconsistencies in formatting and as the different tools produce different outputs and researchers have to implement their own methods of collecting those features.

Certain tools have been created to alleviate this problem but they tend to be resource intensive and have limited portability. Taking this issues into consideration, we propose a simple and versatile Ultimate Feature Extractor tool that is designed to gather and present Android app features within the context of mobile security research.

Discipline of research: Mobile information and security

The tool is co authored by my colleague, Haoran Xia. We have written separate dissertations for the tool with a focus on our own individual contributions towards the tool.

Contents

1 Introduction **3**

2 Overview of pre-existing tools **4**
2.1 AndroPyTool overview 4
2.2 Apktool overview 5

3 Tools under utilization **6**
3.1 Androguard overview 6
3.2 Dex2Jar with Fernflower overview 6
3.3 Androguard versus Dex2Jar with Fernflower for source code retrieval 7

4 Features of interest **8**
4.1 Task division 8
4.2 Contextual features 9
4.2.1 Google Play 9
4.2.2 Virus scanning web services 10
4.3 Technical features 10
4.3.1 String constants and obfuscations 10
4.3.2 API methods 10

5 Implementation details **11**
5.1 Extraction of contextual features 11
5.2 Extraction of technical features 12
5.2.1 String constants and obfuscations 13
5.2.2 API methods 13
5.3 Output and formatting 14
5.3.1 Contextual features output structure 14
5.3.2 Technical feature output structure 15

6 Evaluation **17**
6.1 Evaluation of reports from virus scanning services 17
6.2 Evaluation of the performance of the components in contextual feature extraction pipeline 19
6.3 Evaluation of the performance of the components in source code feature extraction pipeline 21

7 Results and analysis **22**
7.1 Contextual features results and analysis 22
7.1.1 Google play contextual features results and analysis 22
7.1.2 Virus scanning contextual features results and analysis 25
7.2 Source-code features results and analysis 27

8 Conclusion **28**

9 Future work **29**

1 Introduction

Android is an open source mobile operating system that has grown in popularity over the years since its introduction and launch in 2007-8. At the current moment, Android sits as the most prominent operating systems of mobile devices around the world. Besides its main presence in the mobile phones market it is also prominent on smart devices such as watches, cars, glasses, TVs and even home appliances.

As this operating system grew in its popularity, the greater numbers of software applications designed for this prominent operating system. To distribute and service these software applications, digital stores such as Google Play, Samsung Galaxy Store or Huawei App Stores were introduced. In addition, there was also the introduction of several third party digital store alternatives that offered several mobile apps for download and use.

However, with these greater numbers of software applications developed for the Android operating system, the number of mobile apps that took advantage of exploits and attacks to cause harm to several end-users also rose in number. Up to the present time, anti-malware applications are constantly on task to combat software apps that are distributed with malicious features and vulnerable components.

Therefore, there was and still is a need for analyzing and detecting Android mobile applications for potential vulnerabilities and attacks directed towards end users. Particularly, for detecting zero-day exploits which are exploits that are publicly known or exploits that are unaddressed by the parties that are responsible for constraining the vulnerability.

To detect and find these exploits mobiles apps have to be examined and analyzed thoroughly for vulnerabilities and behaviours that are malicious in nature. Using a variety of data extraction and analysis tools, there are several studies examining and analyzing a variety of different mobile appellations in order to detect and classify applications based on their vulnerability and maliciousness. However, even if these several tools analyze different components using different tools, there is a lack of a unified multipurpose tool that would extract and present a structured set of featured data.

We first introduce two subdivisions for the features that we are interested in. These subdivisions are contextual features and technical features. Contextual features are features that are gathered from sources outside the given application and technical features are features that are gathered from the Android applications themselves.

And therefore as part of the development of the tool, in this research paper, we will raise the following research question:

Research question: What is the effect of gathering contextual and technical features on Android applications in the static based security analysis?

Additional sub-questions to answer:

- What consists of the contextual and technical data within the tool?
- What is the performance breakdown of contextual and a subset of the technical components?
- What are some example uses for the collected data in the context to mobile security research?

With these questions in mind, we will first have a quick overview of some of the pre-existing tools that did some work on a similar area.

2 Overview of pre-existing tools

There are several tools to pre-existing frameworks or tools that are designed for the purpose of extracting and presenting features from Android applications. One of these most notable tools is an open-source Python based framework called AndroPyTool [9]. In addition, Apktool is another popular tool that is used to reverse engineer and acquire the source code of a given Android app package which allows for analysis for source code of the mobile application. These tools are further explored on the section 2.1 and 2.2.

2.1 AndroPyTool overview

AndroPyTool is a tool used for obtaining static and dynamic features for a given set of Android applications. The tool itself uses several well-known pre-existing tools such as DriodBox [6], FlowDroid [1], Strace, AndroGuard [2] or VirusTotal analysis. AndroPyTool employs several steps in order to provide a full report on the features of an Android application [9]. These steps include:

1. **Apk filtering**: A step that is used to filter APK files that are invalid or corrupted before analysis.

2. **VirusTotal analysis**: Uses a popular antivirus scanning service to receive a report security malware analysis that is already present in the antivirus databases.

3. **Dataset partitioning**: This step separates samples into malware and benign sets based on a report from VirusTotal and a threshold specified by the user. a user.

4. **FlowDroid execution**: uses the FlowDroid open-source tool used to statically analyze data flows within Android applications.

5. **FlowDroid result processing**: this step allows for the processing of the results acquired from the FlowDroid execution step.

6. **DroidBox execution**: The application is analyzed for features that can only be acquired from run-time or execution of the app.

7. **Feature extraction**: The final step involves aggregating the information into common file formats such as comma-separated values (csv), JavaScript Object Notation (JSON) and as a database based on MongoDB.

AndroPytool performs quite well in terms of the extent of the data it gathers which involves pre-static, static and dynamic related features. However, the tool certainly has some shortcomings which is difficult to overcome without a complete overhaul. One of its noticeable drawbacks is that the tool requires several libraries and packages which makes heavy (memory space intensive) and it has limited modularity of these packages.

Additionally, as this tool heavily relies on several other tools the management or installation process tends to be difficult especially in cases where the mode of installation cannot occur using a container system such as Docker. And the container system, Docker, has certain special requirements/privileges that in some cases cannot be easily met. These requirements include not being supported in virtual machines and requiring operating systems such as Windows to be in the Enterprise or Pro versions etc.

Another significant drawback to the tool is that it is based on a Python version (Python 2.7) that is under the End of Life (EOF) status as 2020 and it will no longer receive any official support from the Python software foundation [13]. Furthermore, the source code of the tool lacks modularity which can translate to reduced performance and increased difficulty in testing and maintaining the tool.

2.2 Apktool overview

Apktool is an open-source tool that is used to acquire the source code of Android application via generalized reverse engineering techniques [15]. It is capable of decoding the assets found in an Android application package to nearly their original form. Once this near original form is acquired, several analysis tools particularly focused on analyzing the source code can be employed and used to extract meaningful data on the behaviors of a given Android application.

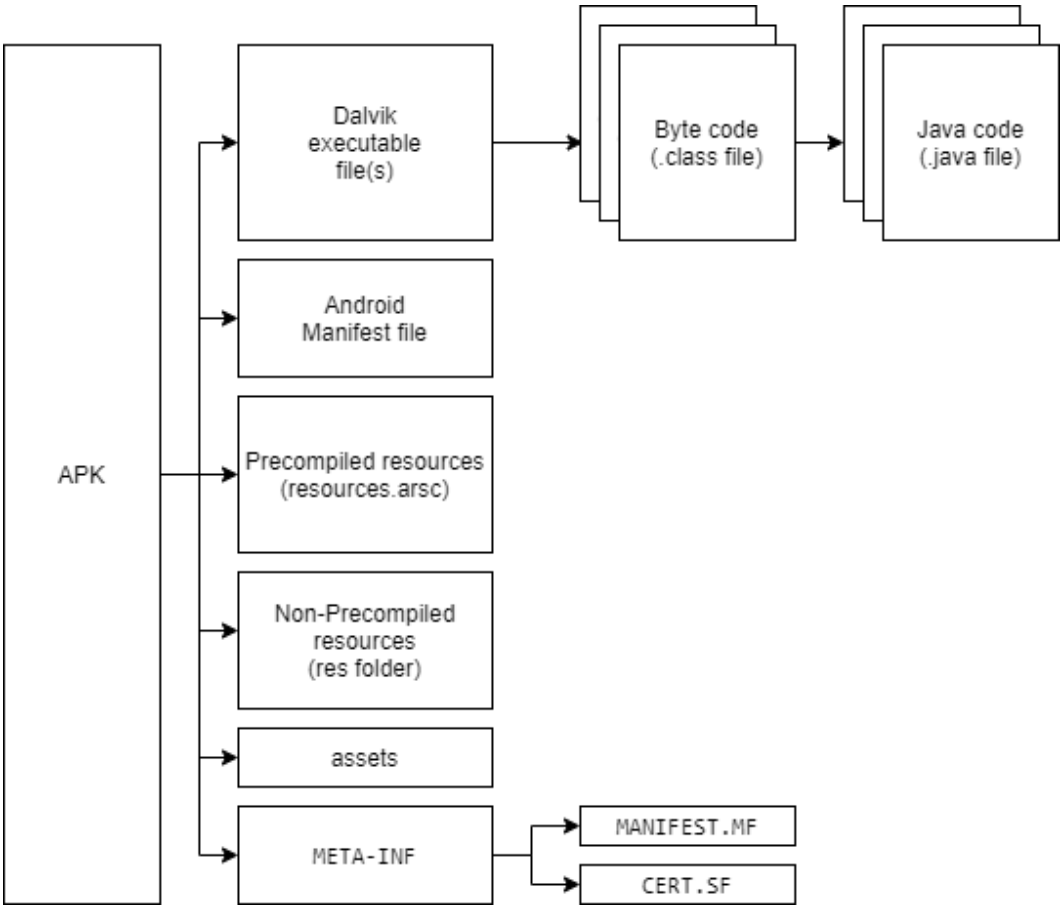


Figure 1: The typical contents of an Android application package (APK)

Figure 1 illustrates the common contents of an Android application package. The common contents include the `resources.arsc` file, the `dex` file or files, the Android manifest file and other resource related files [10]. Apktool features the disassembly of these Android application package components such as:

- **Resources.arsc**: contains the meta-information on the resources involved within an Android application [3]. These meta-information include static contents such UI related information such as the layouts and also the attributes for these layout components, images or bitmaps and localization information of the app.
- **Classes.dex**: contains all the Java class files where the primary Java source code lies. These Java class files consist of compiled bytecodes that are converted to Dalvik-executable dex files.
- **AndroidManifest.xml**: is an essential file present within all Android applications that describes meta-information about the application it is presented in. The man-

ifest file is required to include a unique application identifier (package name), the components of the application (activities, services, broadcast receivers and content providers), the permissions and finally the hardware and software features that the application requires.

3 Tools under utilization

Taking the above mentioned tools into consideration, the project uses three main tools in particular Androguard [2], Dex2jar [11] and Fernflower [4] to get features from Android application source code.

3.1 Androguard overview

Androguard is a Python based Android application package reverse engineering tool that provides run-time analysis objects that can be used for feature extraction and processing. This tool presents a variety of interfaces or Python API calls for getting information about an Android application package (APK).

To acquire the sets of information/features from an Android application, Androguard has three main designations or components which are:

- The `a-object` (**APK** object): contains all the information about an APK file. These include the package name, permissions, Android manifest file, the app resources and more.
- The `d-object` (**Array of DalvikVMFormat**): contains the array representation of the Dalvik executable files that are found within the APK.
- The `dx-object` (**Analysis** object): contains special classes and is designed to contain information about the multiple Dalvik executable files within the APK package. This object is prefered to be in use over the `d-object` for APKs that contain multiple DEX files. This object is used to get the Java source code for the given Android application. However, this retrieval of the Java source code is not perfect and has some limitations to which another tool described in section 3.2 offers to overcome.

3.2 Dex2Jar with Fernflower overview

As Androguard exhibits some limitations in terms of retrieving the Java source code from Android applications. These limitations include inability to convert certain Java source files and to retrieve import statements. The Dalvik executable files that Androguard fails to analyze usually feature Java source codes that are heavily obfuscated. These limitations of Androguard lead to the implementation or addition of an alternative pipeline for retrieving the source code related features. This pipeline involves two reverse engineering tools namely Dex2Jar and Fernflower.

Dex2Jar is an open-source tool that is designed to convert Dalvik executable files into files into Java archives (`JAR` package format files) [11]. This format is in turn used by another open-source tool, Fernflower [4] developed by Jetbrains to decompile Java archive (`JAR`) files into their original Java Source Code form.

However, it is noted that the pipeline of getting the source code using Dex2Jar with Fernflower is not perfect and has definite limitations. A combination of Dex2Jar and Fernflower with Androguard enables us to realize a combination of the advantages of the two approaches. And these advantages and disadvantages are explored in the following Section 3.3.

3.3 Androguard versus Dex2Jar with Fernflower for source code retrieval

Both these methods/tools have their advantages and disadvantages as outlined by Table 1.

	Androguard	**Dex2Jar with Fernflower**
Speed	Fast	Slower
Import statements	Unavailable	Available
Accuracy	Limited	Very good
Specialized functions or run-time objects	Present	Absent

Table 1: The advantages and disadvantages of Androguard against Fernflower with Dex2Jar for source code retrieval

Androguard has an advantage over Fernflower in terms of speed however Fernflower has the advantage over how accurately the APK is reverse engineered to the original Java source code. This reduced accuracy is best exemplified by how Androguard presents inline imports rather than presenting the import statements themselves. The use of inline statements makes it very difficult or otherwise impossible to retrieve import statements via common retrieval methods such as searching or parsing.

Furthermore, since Androguard is a tool that offers Python run-time analysis objects for Android applications, getting the components via Androguard's specialized functions is very straightforward.

As an example, Androguard provides specialized `strings` object within its analysis `dx` object that contains all the string constants present with in the given Android application. In addition, these string constants are retrieved using specialized objects called string analysis objects which provide additional information for each string constant present within the given Android application.

One of the additional information provided is the cross reference graphs (`XREF`s) that would provide information on which Java methods or functions inside the given Android application source code are utilizing the string constants.

4 Features of interest

Based on the interests of papers studying Android application package files [17], the feature of interest for this paper are outlined and explored as follows:

- **Contextual features**: includes general information that gives contextual information about a given application. More specifically, contextual related information involve the title, reviews, description, monetization, etc. of an application from Google Play [8] [14] and additionally, information on vulnerability or maliciousness from antivirus scanning web services such as VirusTotal and others are included.
- **Technical features**: includes technical and in depth information on a given Android application. It includes string constants and obfuscation, API calls or methods, operational code (opcodes), name obfuscation, etc.

This paper focuses on the contextual features and a certain subset of technical or source code features. The technical features that this paper focuses on are string constants and their obfuscation along with API methods which are outlined on the Section 4.3. The tool that this paper is based on is not limited to technical or source code features presented in this paper, the other features are explored by a different author (Haoran Xia) who worked on those features.

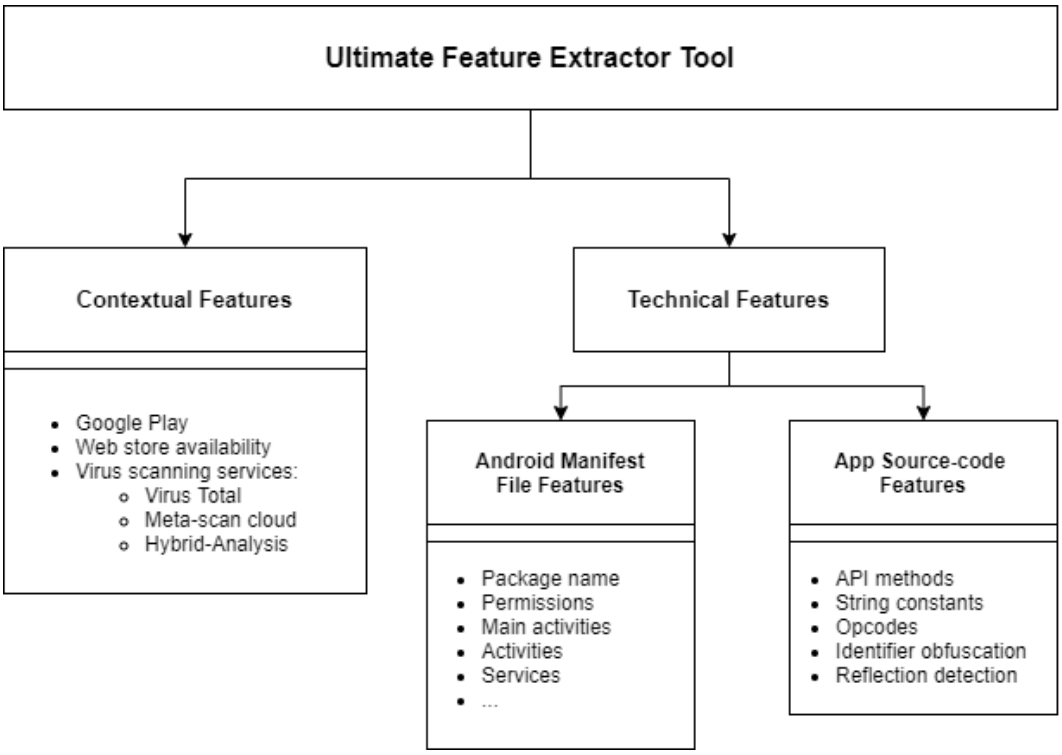


Figure 2: The architecture of the Ultimate Feature Extractor tool

4.1 Task division

- **Yona Moreda**: worked on Contextual features and a small subset of Technical features (API methods and Strings)
- **Haoran Xia**: worked on Android Manifest file features and Technical features (Opcodes, Identifier obfuscation and Reflection detection)

4.2 Contextual features

When analyzing an Android application, it is often useful to retrieve and analyze the contextual information of the application to have an insight to the behaviour or supposed purpose of the application. Contextual features involve several services to retrieve and present meaningful contextual data for a given application.

4.2.1 Google Play

Google Play is the official digital library web store that serves several Android applications for many mobile devices. While Google play distributes and sells Android applications, it does so by providing valuable contextual information [8] that includes meaningful attributes as outlined below.

- **Title and package name**: are used to identify and find the given application on Google play. The package name in particular is uniquely assigned to each application that is served by Google Play.
- **Category**: provides contextual information to which associations can be made to the expected behavior of an application in the given category.
- **Release date and date of update**: provides the time the application is originally released on the Google Play store and also information on the date the application was changed or updated. This information is helpful for tracking changes for giving additional data in cases of application repackaging.
- **Description**: presents descriptive information of the application. This element provides the application description as outlined by the developers of the application. This is usually the first piece of text that the users read before installing the application and it provides valuable context to the expected behavior of the applications.
- **Developer information**: consists of the developer name, address, email, unique developer identification number and their website.
- **Monetization**: includes information about how the pricing of the application or the other means of using the application to generate revenue. This information includes the price of the application, the currency, whether or not the application is supported using in-app purchases, the range of prices for the in-app purchases and the presence of in-app advertisements.
- **User/Community feedback**: composed of information describing the experience of users when using the application. This information includes the top comments left by the users, the average rating, the total number of reviews and installs, the distribution of the ratings from 1 to 5 stars (as a histogram), if the application is part of the editor's choice and finally the content rating for enforcing parental control.
- **Privacy policy**: provides information on how user data is gathered, used and managed by the developer of the application. Contextual information regarding issues of privacy should be addressed in the privacy policy.
- **Size**: provides information on the size of the application.
- **Screenshots and Video**: presents demonstrative screenshots or video recordings of the application in action in other cases it presents images of the expected features of the application. The developers are allowed to make a variety of design choices regarding the contents of the screenshots and videos as long as the contents are within the technical standards of Google play. This component can often heavily be part of the promotional component of the application.

4.2.2 Virus scanning web services

Virus scanning web services are services that serve reports of malicious behavior or vulnerability by aggregating and using a variety of virus scanning engines or services. These services provide contextual information on whether a given application has already been flagged or recognized as a malicious application. The services that are in use in this paper include VirusTotal, OPSWAT MetaDefender cloud and Hybrid-Analysis.

- **VirusTotal**: aggregates over 60 virus scanning products or engines to detect malicious behavior. It was acquired by Google in 2007 and it is an valuable web service for users that are interested in getting reports on the malicious reputation of applications.
- **OPSWAT MetaDefender cloud**: offers over 30 antivirus scanning engines via the cloud. It serves as a complementary tool to the reports provided by VirusTotal.
- **Hybrid-Analysis**: provides an analysis of threats for a given Android application.

Overall, with the combination of these virus scanning services that aggregate other virus scanners, it is possible to acquire a near exhaustive contextual information regarding its maliciousness of a given Android application.

4.3 Technical features

Besides taking account of the contextual information surrounding an application, it is important that the more in depth technical features are extracted and placed under scrutiny. Several papers related to security use technical features that involve the Android manifest file and APK source code components to evaluate the security related features of an Android application [17]. In the following sections, Section 4.3.1 and Section 5.2.2, the overview of the technical components are established and detailed.

4.3.1 String constants and obfuscations

String constants or string literals are constants that store a fixed sequence of characters. In Android applications that are based on Java, these items are surrounded with double quotation marks and they are one of the well-known methods that are used to obfuscate snippets of code. Malicious applications can store literal encrypted string literals with methods which are activated at run time by using Java's Reflection API. String literals used this way would evade detection of common static inspections of the source code [5].

String constants as a feature are also useful in other aspects. They often contain the file paths, URLs, IP addresses and other similar sets of data that is used by the application for possible malignant reasons [18]. Although the string literals containing nefarious items like URLs or IP addresses are likely to have been obfuscated or decomposed into fragments to avoid detection, its possible to employ various detection and analysis methods to account for such cases and study the behavior of a given application.

Several papers have used and analyzed string constants to assess malicious behaviour of applications and those studies were the motivations behind examining and string constants in the project [17].

4.3.2 API methods

API methods or calls outlines how an Android application interacts with the Android framework. API calls provide an interface for interacting with the Android device components. As an example, these methods include calls made to the camera device to capture an image,

calls made to the Android SMS manager to access contacts or calls made to interact with the Bluetooth interface and several other more. These features can be used to flag certain apps as suspicious using state of the art analysis tools [12]. The analysis of the API methods component has the potential to be resourceful in the process of detecting and deterring zero-day exploits or vulnerabilities.

Certain applications that seldom request certain API calls that contain sensitive data are resourceful in revealing hidden malicious behavior. This component works with regards to the contextual information that details the intended or advertised purpose of the application that would be compared against the underlying implementation of the application.

It is noted that API calls are not accessible to the Android application unless the user explicitly grants permissions via the Android permissions API. However, the API methods provide a more in depth insight to what exact methods are used for the requested permissions and help assess the threat level of different levels of API calls.

5 Implementation details

In this section, the implementation details are outlined for the features for the contextual and a subset of technical features. These features have already been described and discussed in the Section 4.2 and 4.3 and in the following Section 5.2 the implementation details are discussed.

5.1 Extraction of contextual features

To retrieve and extract the contextual features data from Google play and Virus scanning web services, the services are queried and the results are processed, organized and formatted. To interact with these services, we use the different publicly available API endpoints for each required task.

However, with one exception for the way the data is gathered from Google Play. The data gathered from Google Play is collected through the methods of web scraping as there are no alternative methods for querying and gathering information from Google Play. Google Play does not officially provide a publicly available API for the information it serves on its applications. This method of gathering the data through web scraping is quite reliable and our tool uses two distinct Python based open-source libraries for collecting data from Google Play.

The involved steps for gathering contextual component is shown as follows:

1. Acquire data from Google Play web based scrapers
    1.1. **Acquire data from play-scraper**: `play-scraper` is a Python based library that provides an interface for gathering information from Google Play about an App. The package id of the application is given as a parameter.
    1.2. **Acquire data from google-play-scraper**: `google-play-scraper` is a secondary Python based library that provides additional information provided besides play-scraper.
2. Acquire data from VirusTotal API
    2.1. **Request via file hash**: `GET`-request to VirusTotal public API using `SHA256` digest of the file to gather contextual reports from virus scanning services.

2.2. **Request via file upload**: if enabled, it would allow request for a report by uploading the APK file itself granted the report cannot be received via a request through the file hash. Even when enabled the tool will not upload the file if the report can be gathered through a request with a file hash (for increased efficiency). Furthermore, this file upload is non-blocking for an execution with a batch collection of tasks.

3. Acquire data from Meta-scan cloud API

   3.1. **Request via file hash**: `GET`-request to Meta-scan cloud API using `SHA256` digest of the file.

4. Hybrid-analysis API

   4.1. **Request via file hash**: `GET`-request to Hybrid-analysis API using `SHA256` digest of the file.

5. Acquire data on App store availability

   5.1. **Request via package name**: `GET`-requests to selected few web-based App stores to check if the application is available for download.

The collected data is finally written to the output file, details on the output of the application is discussed in Section 5.3.

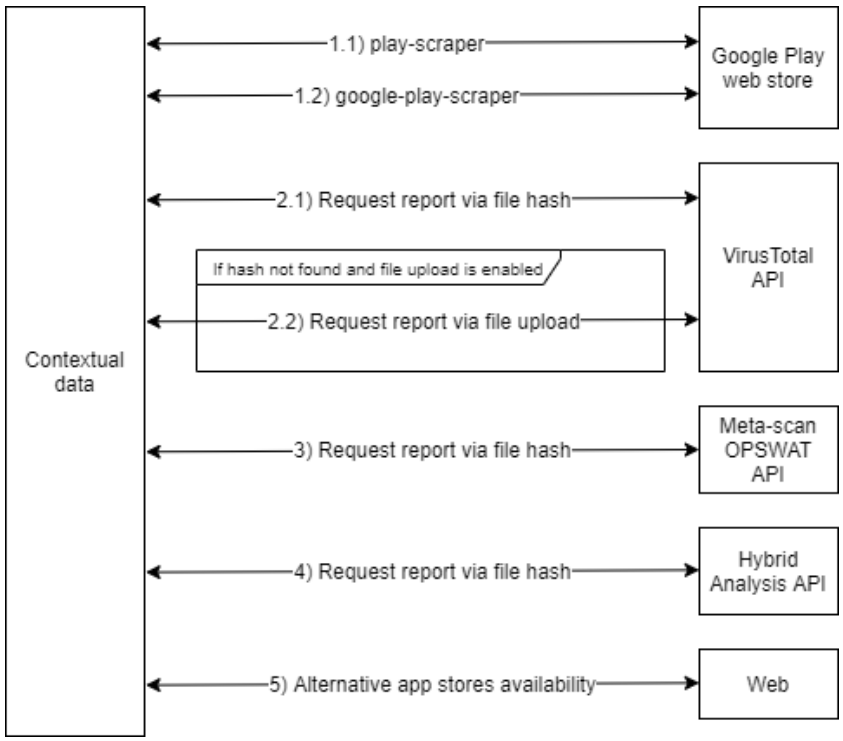


Figure 3: The steps involved in contextual feature extraction

5.2 Extraction of technical features

The extraction of the technical features involve analyzing the Android Application Package (APK) through the use of existing tools. The tool that is in use in order to extract technical features is Androguard and its overview is outlined in Section 3.1. Androguard provides

specialized functions that return Python based run time objects that present technical information regarding an Android application.

5.2.1 String constants and obfuscations

Given an Android application package using Android, the following steps are executed to retrieve the string constants data set:

- 1st - Dalvik Analysis (`dx`) object is created.
- 2nd - The string analysis objects are acquired from the analysis (`dx`) object.
- 3rd - For each string analysis object, check whether it is obfuscated by checking if it is obfuscated via common methods. If it is obfuscated, it is counted as an obfuscated string and aggregated into a list. Strings that indicate snippets of executable Java code or strings that have a different encoding are counted as possible string obfuscated items.
- 4th - Present the list of string literals in the application along with a list of strings that have been identified as obfuscated. The tally of possibly obfuscated strings is also presented.

The method of checking whether a string is obfuscated or encrypted has definitive limitations as it checks for common encoding methods or whether uncommon characters are present. More sophisticated encryption methods are designed to be quite difficult to detect and designed to be indistinguishable from random text of data.

The fact that encryption methods are made so that they are indistinguishable from random text of data offers the possibility of detecting deliberate obfuscation. As encrypted pieces of text are designed to have a very uniform distribution of alphabetical letters while the English text has patterns of favoring the frequency of some letters using this difference as a factor it is possible to develop a heuristic that detects encrypted strings.

However, this pattern only emerges for pieces of texts or strings that have long lengths and as the detection method is based on general heuristics the reliability of the detection is limited. As the implemented obfuscation method has limitations, the tool offers the complete list of string literals along with the list of obfuscated strings. With the component based design, the user can enable the retrieval of only the obfuscated strings without collecting all the string literals.

Furthermore, the string obfuscation heuristic uses a simple set of patterns shown in Listing 1 to identify possible executable snippets of code within the string constants. These sets of patterns are parameterized as part of a configuration file and the user is free to provide their own set of patterns for detecting executable codes within string literals.

Listing 1: Configuration file for customizing string obfuscation patterns

```
[String_Obfuscation_Sentinel_List]
sentinels = ["{", ";", "void", "[", "if (", "while(", "for("]
```

5.2.2 API methods

API methods consist of methods that are defined within the Android development framework. These methods are accessed using an analysis (`dx`) object created by Androguard. These methods are under objects that are called `ExternaClass`. An `ExternalClass` are classes that are not defined APK, and are called external for this reason.

The steps involved in acquiring API method is as follows:

1st - Dalvik Analysis (`dx`) object is created.

2nd - The external class analysis objects are acquired from the analysis (`dx`) object.

3rd - From the external class analysis objects, the method analysis objects are acquired.

4th - From the method analysis objects, both the method and class name are combined together as a unit with a form `<class-name>::<method-name>` and are stored in terms as a histogram of API methods for the given application. This formatting and organization is explored in detail in Section 5.3.

Overall, these provided methods (and classes) along with their frequency give an insight to the inner workings of a given Android Application.

5.3 Output and formatting

A careful and deliberate consideration has been given into making sure a standard formatting is provided as an output for each component. There are two possible offerings that the tool provides and they are comma-separated values (CSV) and the Java-Script Object Notation (JSON).

The data is kept inside a Python dictionary and it is written into the CSV and JSON formats in the final step of a feature component. The field `package-name` or the package-id is used as the primary key for when the CSV or JSON files are written into.

5.3.1 Contextual features output structure

For the features extracted from Google Play, the structure is set so that for each field described in the Google Play store, there is a corresponding header field for CSV and key for JSON. A subset of formatted contextual data from Google Play is seen in Listing 2.

Listing 2: Snippet of contextual features sample data (Google Play)

```
{
    "adSupported": true,
    "androidVersion": "4.0",
    "category": ["TOOLS"],
    "comments": [
       "Pretty perfect for me. I searched for a easy app to show me a straig ...",
        "Excellent Compass... Can we please have a Setting to set the Lat / ... ",
        "Really really good. Works very smoothly and has a good balance of fe...",
        "A very nice app, it does all the jobs it promised to do. But the GPS...",
        ...
    ],
    "containsAds": true,
    "contentRatingDescription": null,
    "content_rating": [ "Everyone" ],
    "currency": "USD",
    "current_version": "1.55",
    "description": "Use this app to:- Save your current GPS ..."
    "developer": "Evgeni Ganchev",
    "developer_address": "Bulgaria, Plovdiv, ...",
    "installs": "1,000,000+",
    "package-name": "com.gpsnav.evo.gps2",
    "privacyPolicy": "https://sites.google.com/view/gpscompass/gps-compass-e...",
    "released": "Dec 29, 2014",
    "reviews": 4546,
    "score": 4.154867,
    ...
}
```

In the case of the results from the virus scanning services, there are three fields that provide the number of antivirus engines that have identified a threat and in addition there are three fields that present the list of the antivirus engines that identified the application as a threat.

Listing 3: Snippet of contextual features sample data (virus scanning services)

```
{
    ...
    "HA_threat_score": null,
    "HA_positives": null,
    "HA_positives_list": null,
    "opswat_result": "Infected",
    "opswat_positives_list": ["BitDefender", "Commtouch", "Emsisoft", ...],
    "vt_positives": 18,
    "vt_positives_list": [ "CAT-QuickHeal", "McAfee", "Trustlook", ...],
    ...
}
```

Output entry format for gathered virus scan reports:
`"<scanner>_positives": <count-of-positives>`
`"<scanner>_positives_list": <list-of-positive-antivirus-scanners>`

The `<scanner>_positives` fields describe the number of antivirus engines that have identified the application as a threat. The token `<scanner>` is `HA` for Hybrid-Analysis, `opswat` for OPSWAT's meta-scan and `vt` for VirusTotal. The same notation is utilized for CSV formatting where `HA_positives`, `HA_positives_list` and the rest of the fields are described as CSV headers.

5.3.2 Technical feature output structure

The organization and presentation of the technical features follows simple approach and is essentially similar to approach of the contextual features. The data is stored in Python dictionary object and later written in a CSV and/or JSON file.

For API methods, a frequency distribution (histogram) is created and this distribution is saved as a field under `api-methods` for each package identification name or `package-name`.

Output entry format for API Methods:
`"<classpath><classname>[$innerclass]::<methodname>": <frequency>`

Listing 4: Example of source code features sample data for API methods

```
{
    ...
    "Landroid/hardware/Camera::open": 1,
    "Landroid/hardware/Camera::release": 1,
    "Landroid/widget/TextView::getAnimation": 1,
    "Landroid/hardware/Camera::setOneShotPreviewCallback": 1,
    "Landroid/hardware/Camera::takePicture": 1,
    "Landroid/hardware/Camera::getCameraInfo": 1,
    "Landroid/hardware/Camera$Parameters::setFlashMode": 1,
    "Landroid/hardware/Camera$Parameters::setFocusAreas": 1,
    "Landroid/hardware/Camera$Parameters::setFocusMode": 1,
    "Landroid/hardware/Camera$Parameters::setMeteringAreas": 1,
    ...
}
```

In the example data (Listing 4), the application in question uses the Android API framework to access hardware features and make API calls such as `takePicture` or `getCameraInfo` to access the camera hardware device and capture images. The frequency of the API calls is also provided alongside each API method. It is shown that the API methods `getAnimation()`, `setOneShotPreviewCallback()`, `takePicture()` and `getCameraInfo()` are called only once for the given random application sample.

In the case of string literal related features, when string constants component is enabled, the list of string constants and string obfuscations are collected and a tally of possible string obfuscations is recorded and presented. In the output, for each unique `package-name` of an application, the list of all its string literals (`all-string-constants`), obfuscated strings (`possible_obfus_strings`) and the obfuscation tally (`possible_str_obfs_cnt`) is presented. An example of the string literals component is shown in Listing 5.

Listing 5: Example of source code features sample data for String literals and obfuscation


```
      "app_details": [
          {
              "all-string-constants": [
                  "BookMark.db",
                  "BOOKMARK_TITLE",
                  "BOOKMARK_CONTENT",
                  "BOOKMARK_INDEX",
                  ...
              ],
              "package-name": "org.shofwatuna.magazine.AOVIDBXMHBCQYFOP",
              "possible_obfus_strings": [
                  "onDoubleShow();",
                  "onSingleShow();",
                  "mShowing",
                  "onDoubleShow();this.showDialog();",
                  "onSingleShow();this.showDialog();",
                  "\u91ca\u653e\u5185\u5b58",
                  ...
              ],
              "possible_str_obfs_cnt": 52
          }
      ]
  }
```


This possible string obfuscation counter is a coarse probability based counter for string obfuscations as detecting string obfuscation at statically significant accuracy is impossible as string obfuscation can have unlimited ways of being generated. This limitation is the reasoning behind providing the string constants along with a count of possible string constants. Any generalized string obfuscation detector will have limitations in detecting strings that are obfuscated.

As the API methods of several applications are gathered and recorded in terms of frequency to understand to observe the behaviors of the applications, attackers use alternative methods such as string obfuscations techniques to conceal their moves or behaviors.

Collecting string constants allows the user (researcher) to have the ability to directly work with the data itself and not be limited to the list of string obfuscations that out methods detect and present.

6 Evaluation

The tool is designed in such a way that each of the above discussed components and subcomponents to be turned on or off for an execution. An execution involves two modes namely batch mode or single APK analysis. There is not a major underlying difference between the two modes, the only difference being the parameter for batch execution is a path to a folder, while single execution uses a path to an application (APK). For this section, in order to evaluate and stress test the tool, the batch execution mode is put into use.

VirusTotal offers a range of virus scanning engines to provide a report on malicious behaviors of given files or applications. However, this scanning service has a restrictive quota on the quantities of reports that can be retrieved from the service, which also holds true for most alternative virus scanning services. To overcome the problem and offer some additional information on the contextual information regarding an Android application, virus scanning services such as Meta-Defender's OPSWAT and Hybrid Analysis are featured besides VirusTotal.

These virus scanners offer more relaxed quotas and more importantly offer redundancy to strengthen the information regarding the malicious behavior. In the following section 6.1, we explore the extent to which this extra information offers value and we explore the performance of these components with respect to the overall performance of the contextual component. We used a data set that consists of 3144 malware and 1543 benign sample APKs and this data set was generously provided to us by our supervisor, Fadi Mohsen, PhD.

6.1 Evaluation of reports from virus scanning services

To evaluate the featured virus report providing services, a malware sample that consists of 3144 APKs was run with the VirusTotal and OPSWAT web scanner components enabled.

The strict quota from VirusTotal sets the maximum allowed requests to be 4 requests per minute. This set quota can be lifted if the tool is provided with an API key that is from VirusTotal premium accounts. However, for cases when a premium key is not or cannot be acquired, the redundancy offered by the report provided alternative services such as OPSWAT MetaDefender alleviates the problem and allows most of the malware components to receive antivirus reports.

When the virus scan report for an APK is required, the virus scanning web services are queried via the hash digest of the APK. However, this report might be absent for various reasons such the hash being not present in the database of the scanning services, or the strict quotas set by the virus scanners to query their service.

Changing the tool to accommodate for these quotas would result in a large performance overhead and completely disregarding tools with strict quotas would limit the quality of the virus detection contextual component and thus by utilizing a mix of virus scanning services (VirusTotal and OPSWAT with the addition of Hybrid-analysis) a much more complete and more refined reports for a given APK sample.

For a malware sample set that consists of 3144 APKs, the total number of absent or present reports are counted and reported in Table 2.

	OPSWAT absent	OPSWAT present	Sum
VirusTotal absent	1115	1089	2204
VirusTotal present	489	451	940
Sum	1604	1540	3144

(a) Presence or absence of virus scan reports

	OPSWAT absent (%)	OPSWAT present (%)	Sum (%)
VirusTotal absent (%)	35.46	34.64	70.10
VirusTotal present (%)	15.55	14.34	29.90
Sum (%)	51.02	48.98	100.00

(b) Presence or absence of virus scan reports (%)

Table 2: Amount of present/absent reports from antivirus services (3144 malware APKs)

As shown in Table 2, by offering both VirusTotal and OPSWAT as virus scanning services simultaneously, it is possible to achieve greater quantities of reports for a large sample set that contains several APKs. For a given APK within a large sample set, a report from OPSWAT is present about 50% of the time on average and in the case of VirusTotal around 30% of the time.

Overall, for a malware sample set of 3144 APKs, it is measured that virus scan reports from VirusTotal and OPSWAT is absent simultaneously from both services at about 35.46% of the time making it possible to acquire at least a single virus scan report in a large sample set to happen 64.54% of the time.

Using both VirusTotal and OPSWAT simultaneously provides an improvement in the absence/presence of reports of 116% over simply using VirusTotal alone and a 30% improvement over simply using OPSWAT alone as an average for a malware sample set (3144 APKs). It should be noted that as these rates are applied for publicly available free versions of both services with no requirements monetary related requirements to utilize the tool.

Furthermore, as the database that VirusTotal and OPSWAT utilize is ever growing and improving the presence of reports and reports from these services will be available at greater rates than the aforementioned rate of 65%.

An additional virus scanning service, Hybrid-analysis, provides additional information and complements the reports from the virus scanning services discussed prior. All these additions or mixes are implemented in such a way that the gathered features are as comprehensive as possible with a suitable performance that can accommodate large sets of APKs.

For samples that have a report from both VirusTotal and OPSWAT consisting of 14.34% of the cases, it is possible to examine the results of these reports to observe the rate at which the reports from VirusTotal and OPSWAT provide information that is conflicting or agreeable information. This involves examining cases where a report from VirusTotal might conclude a given APK is infected while the other reports it is threat free.

	OPSWAT Negative	OPSWAT Positive	Sum
VirusTotal Negative	12	1	13
VirusTotal Positive	18	420	438
Sum	30	421	451

(a) Number of positives/negative results

	OPSWAT Negative (%)	OPSWAT Positive (%)	Sum (%)
VirusTotal Negative (%)	2.66	0.22	2.88
VirusTotal Positive (%)	3.99	93.13	97.12
Sum (%)	6.65	93.35	100.00

(b) Percentage of positives/negative results

Conclusions of reports	Ratio (%)
Consistent	95.79
Conflicting	4.21

(c) Conclusion of reports

Table 3: Rates of positive and negative results for a set of malware samples

Table 3 shows the rates of positive or negative results for a presence of malicious behavior in a given application. It can be seen that for the contextual data that is collected for 451 APKs (14.34% of the total malware set, a set for which a report is available from both VirusTotal and OPSWAT), both reports indicated negative results for 12 times (2.66%) and both reported positive results 420 times (93.35%).

Overall, the reports from VirusTotal and OPSWAT is agreeable 95.79% of the time. This is summarised in Table 3c which shows the rate of consistent or conflicting conclusion of reports from VirusTotal against OPSWAT for the used malware sample set.

Now that the use of combining of these virus scanning services is explored, next the performance of the components that make up the contextual component are evaluated in the subsection 6.2.

6.2 Evaluation of the performance of the components in contextual feature extraction pipeline

To gather and extract contextual information for a given set of applications, there are some steps involved which are described in Figure 3. In this subsection, the performance of the contextual component is evaluated by presenting the average time each step takes in the contextual feature extraction.

The performance metrics for the contextual component is measured for a sample set that consists of 316 APKs. Table 4 shows the average and other statistical measurements on the time taken to gather contextual data for the sample set of 316 APKs. The samples exhibit a relatively high variability (**SD**), the Median and the Range are also gathered and presented in the table.

Contextual feature pipeline component	Units in Seconds				
	Average	Median	Range Min	Range Max	SD
Acquiring package ID	0.12	0.08	0.015	1.024	0.14
Acquiring Google Play data	0.49	0.46	0.146	0.973	0.15
Acquiring VT report (hash query)	0.28	0.22	0.175	0.810	0.12
Acquiring OPSWAT report (hash query)	1.02	1.25	0.414	1.419	0.37
Acquiring Hybrid-analysis report (hash query)	0.36	0.28	0.250	0.932	0.19
Checking for app store availability	2.77	4.30	0.347	7.438	2.31
Writing contextual data to output (CSV & JSON)	0.05	0.05	0.001	0.148	0.03
Total average time per APK	**4.98**	**5.90**	**1.348**	**12.744**	**3.31**

Table 4: The performance metrics contextual component (316 APKs)

As seen in Table 4, it takes 4.98 seconds to gather contextual information for a single APK for a batch task consisting of several APKs. On average, this number is slightly lower for tasks in which only a single APK is analyzed instead of a batch of APKs. This reduced time occurs because a batch task makes consecutive query requests to gather contextual information for APKs that has incurs some performance overhead.

The data shown in Table 4 is plotted in relative terms to illustrate the proportion of time that the contextual sub-components take for a run. Speaking proportionally, checking if an app is available in different app stores takes the most amount of time: 2.77 seconds or 55.68% of the total time per APK whereas writing to the output files takes the least amount of time: 0.05 seconds or 0.96% of the total time.

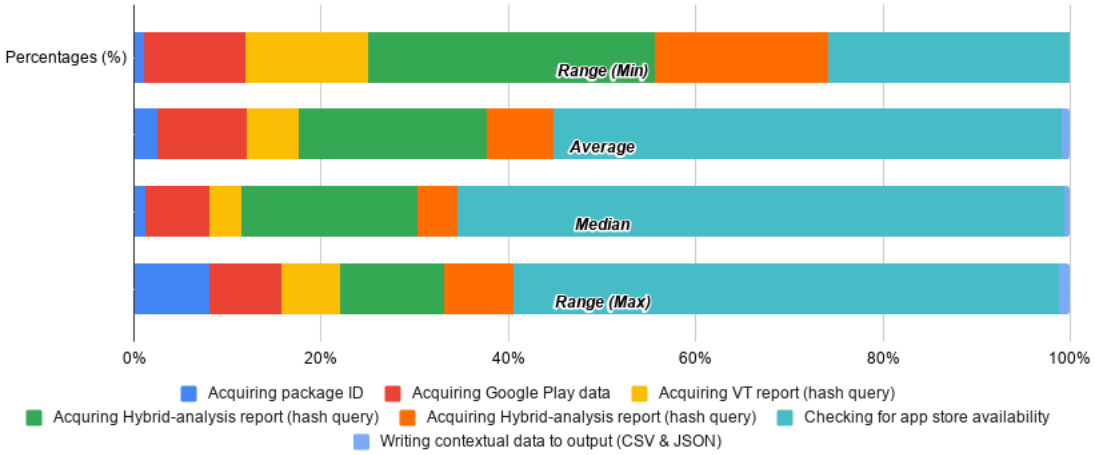


Figure 4: Breakdown of contextual data extraction pipeline (set size: 316 APKs)

It is an important observation that even if the writing to the output files takes the least amount of time, however the time it takes grows linearly as the size of the output file grows and more entries are inserted over time. This effect can be mitigated by partitioning the sample set and the output file to a certain size and handling or compiling the sets as the user wishes in the end.

6.3 Evaluation of the performance of the components in source code feature extraction pipeline

The steps involved in gathering the source code components for string constants and API methods are outlined in Subsection 5.2.1 and 5.2.2. These components are individually bench-marked and the data on some of the performance metrics is gathered and presented in Table 5.

Source-code feature component	Units in Seconds				
	Average	Median	Range Min	Range Max	SD
Creating Androguard APK object	0.128	0.078	0.013	1.109	0.141
Creating Androguard analysis (dx) object	13.141	11.302	0.006	56.908	11.874
Gathering API methods	0.005	0.005	0.000	0.013	0.003
Gathering String constants	0.065	0.048	0.000	0.711	0.078
Writing data to output (CSV)	1.303	1.277	0.049	4.314	0.715
Writing data to output (JSON)	2.279	2.271	0.040	4.643	1.232
Total average time per APK (seconds)	**16.920**	**14.980**	**0.108**	**67.698**	**14.043**

Table 5: The performance metrics of source-code components (Strings constants and API methods only) (316 APKs)

Gathering the string constants and the API methods from the Androguard `dx` analysis object takes 0.065 and 0.005 seconds or 0.381% and 0.028% on average respectively. Creating the `dx` analysis object takes the most amount of time 13.141 seconds or 77.664% of the total time for this particular source-code feature extraction pipeline. Gathering the string constants and the API methods from the `dx` object takes the least amount of time since the creation of the `dx` object involves gathering the string constants and the API methods together and retrieving these items from this analysis object is a trivial task.

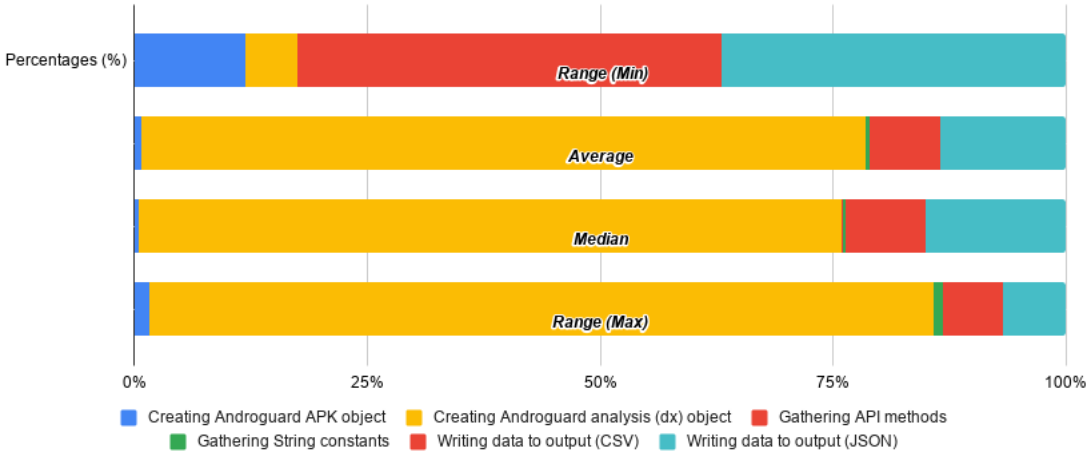


Figure 5: Breakdown of source code data extraction pipeline (set size: 316 APKs)

As the source code related data is gathered over several APKs, in particular for the case of String constants and API methods, the output that is generated becomes fairly sizable over time. This translates to more time being taken by the steps that write to CSV or

JSON. This increase of time consumption by writing to the output files is illustrated in Figure 6.

Figure 6: Writing to CSV and JSON over time in batch task for 158 APKs (API methods and string constants enabled)

This increase in time consumption is also present for the contextual component but at a lesser effect since the data generated by the contextual component is much smaller than this source code component. An approach where the sample set and the output generated is partitioned by the user and only using these components for smaller data sets will prevent extra time from growing or having an effect. In this manner, the user has control to request to gather a more verbose data extraction suited for smaller sample sets or a basic or light data extraction suited for large data sets.

7 Results and analysis

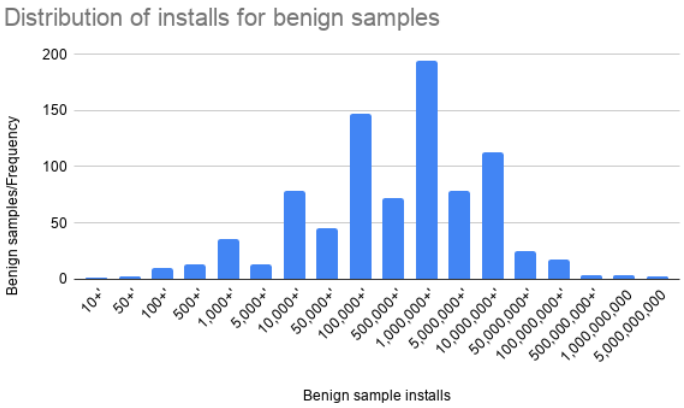
Now the tool itself has been evaluated, in this section, some basic demonstrative analysis is conducted on the results that are gathered from the contextual and source code components. Data is collected for a sample set that consists of 1580 malware APKs and the observations are presented in Section 7.1 and 7.2. All the figures and data are gathered and used the information directly from the output files generated by the Ultimate Feature Extractor tool.

7.1 Contextual features results and analysis

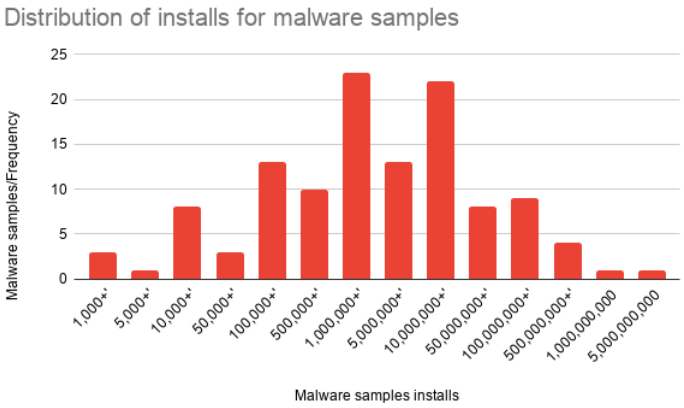
A subset of data gathered from Google Play and the virus scanning services are presented and analyzed moderately in the following subsections of 7.1.1 and 7.1.2.

7.1.1 Google play contextual features results and analysis

The contextual features from Google play as listed in subsection 4.2.1 include the number of installs that an application received. This number of installs provides an insight to the popularity of an Android application with greater installs corresponding to greater popularity. For a sample set consisting of 853 benign and 119 malware APKs, the distribution of installs is illustrated in Figure 7a and 7b.



(a) Distribution of installs for benign samples



(b) Distribution of installs for malware samples

Figure 7: Distribution of installs benign and malware samples

The number of installs for some of the malware applications is large in size, in fact for the dataset illustrated above (Figure 7), the malware APKs are generally prevalent for installs that are greater in size. This observation is more clearly displayed in Figure 8 where the number of installs is presented as in relative terms or as as a ratio to compare the frequency of installs for the sample set of benign versus malware samples.

Figure 8 shows the benign APKs are more prevalent for installs less than 500,000+ and the malware samples are more frequent for installs greater than 500,000+ for the given sample set. Based on the collected data, the malware samples are absent for installs less than 1000+.

Attackers often target popular and prevalent APKs to embed malicious code within the APKs, distribute them after repackaging them as the original harmless application [7]. This sample set illustrates how applications that are popular are targeted. According to the collected data, it is more common to find malware samples that are tied to applications on Google play with installs greater than 1,000,000.

Google play employs some measures to prevent malicious apps from being served on its store but it has limited prevention capabilities for apps that are repackaged with malware [16]. Overall, this contextual information is useful for detecting repackages and illegitimate apps distributed through unpopular means.

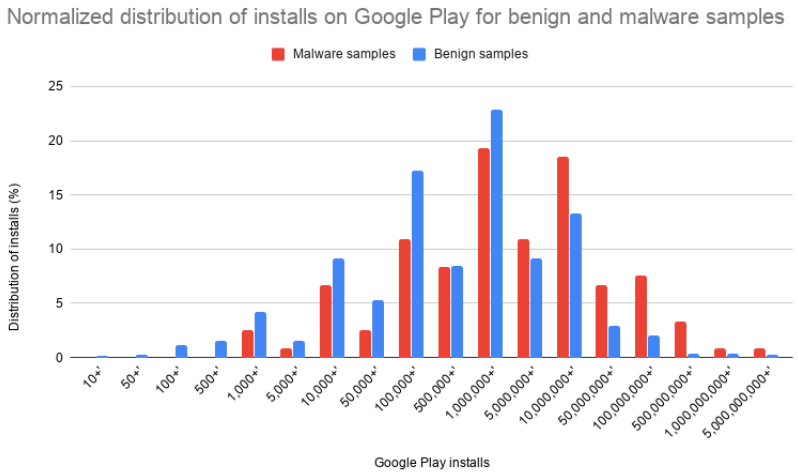


Figure 8: Normalized distribution of installs on Google Play for benign and malware samples

The evidence for repacking also does not end with the number of installs. The user ratings of an application also give an insight to some evidence for repackaging. This is illustrated in Figure 9 where the contextual data gathered from Google Play on the malware and benign samples is normalized and the ratio of the frequency of reviews is compared between the benign and malware samples.

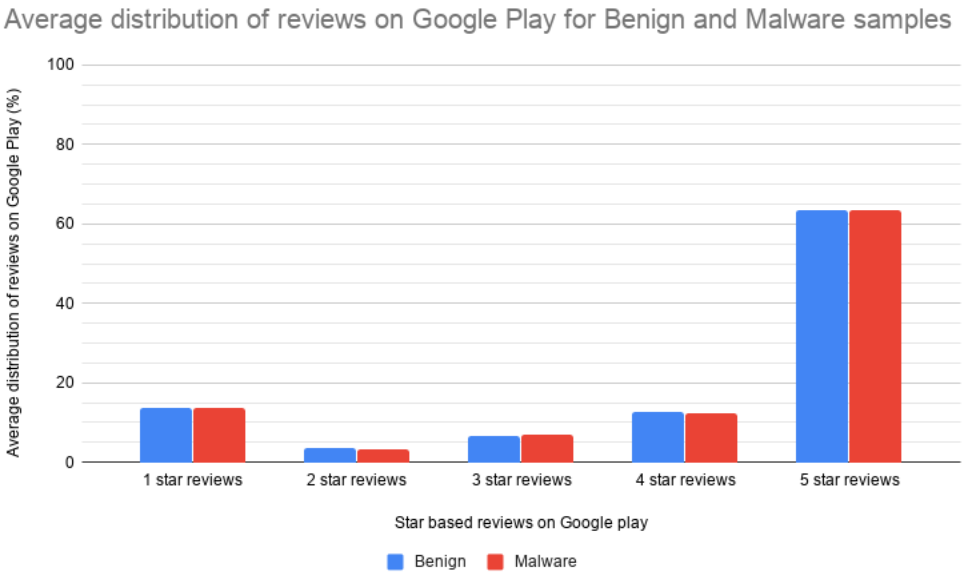


Figure 9: The average distribution of reviews on Google Play for the Benign and Malware samples used (853 and 119 APKs respectively)

As shown in Figure 9, the normalized distribution is nearly identical between the malware and benign samples showing the data gathered from Google play for the malware samples is the contextual data for the genuine applications that are not repackaged with malicious code. The distribution of the reviews is presented in Table 6. It can be seen the difference between reviews in benign and malware samples is less than 0.5% on average for the samples.

Reviews	Average distribution (%)		Average (%)	Standard Deviation	
	Benign	Malware	Delta	Benign	Malware
1 star	13.77	13.62	0.16	11.47	11.02
2 star	3.52	3.34	0.18	2.77	1.92
3 star	6.61	7.04	-0.43	3.60	3.00
4 star	12.66	12.38	0.29	5.86	4.20
5 star	63.43	63.62	-0.19	14.50	12.75

Table 6: Distribution of 5 star reviews for benign and malware samples

The user of the tool could employ more robust comparison or variability measurements to compare and contrast these data sets. For the purposes of this paper, which focuses on the tool itself rather than the processed results only this basic comparison will suffice.

7.1.2 Virus scanning contextual features results and analysis

Contextual features are enabled and data is gathered for a sample set that contains 3144 malware APKs. Out of these requests 888 and 1453 positive reports were aggregated from VirusTotal and OPSWAT meta-scan respectively. These reports are presented as entries in the contextual output files. Each entries contains a list of antivirus engines from the virus scanning services that have identified the given application as a threat.

For the malware samples, the list of positives for each entry is tallied and plotted as a histogram where antivirus engines that frequently identify the malware samples as threats appear in the diagram on the left. Figure 10 and 11 show the sorted list of antivirus scanners from left to right with the antivirus scanners with high frequency of positive detection appearing on the left.

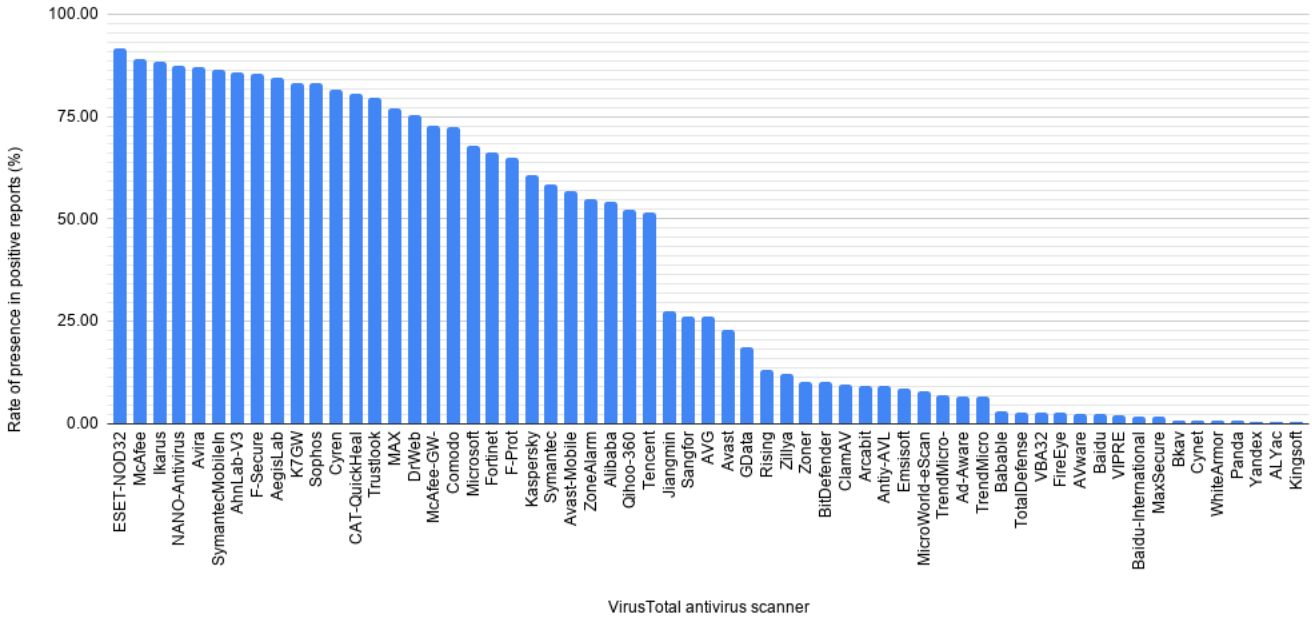


Figure 10: The shares of positive results of VirusTotal scanners from 888 positive reports

Virus scanners `ESET-NOD32`, `McAfee`, `Ikarus`, `NANO` and `Avira` have successfully and frequently identified most of the malware apps present in the sample whereas the virus scanners `WhiteArmor`, `Panda`, `Yandex`, `ALYac` and `KingSoft` identified the malware samples at poorer rates. For the reports received from OPSWAT, `BitDefender`, `NANOAV` and `Ikarus` achieved relatively high rates of positive detection from the collected reports while `F-prot`, `McAfee` and `Windows Defender` reported positive results at a relatively poorer rate.

These figures provide a coarse estimate on the performance of the virus scanning engines in identifying threats. Furthermore, these figures illustrate that the reports from well performing virus scanning engines are close in number and they reinforce the accuracy or the conclusion of the report for a given sample APK.

From the 888 reports collected from VirusTotal, the best performing 28 antivirus scanners have given positive results 450 times or higher (around 50% of the 888 positive reports). And for the 1453 positive reports from OPSWAT, the 6 most successful antivirus scanners contributed to the positive reporting at around 750 times or higher (around 50% of the 1453 positive reports).

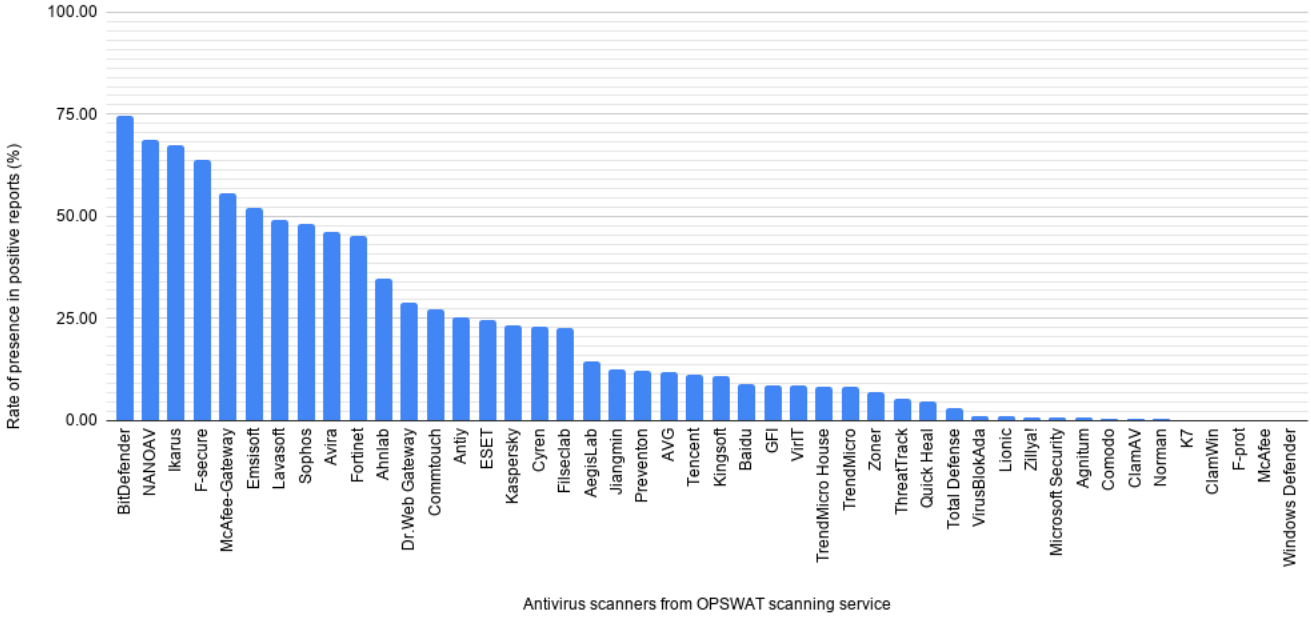


Figure 11: The shares of positive results of OPSWAT scanners from 1453 positive reports

From this figures, its clear to see on average, the reports generated from VirusTotal consist of longer lists antivirus scanners that conclude a positive result on a malware sample in contrast to the results from OPSWAT with shorter lists positive scanners on average.

Moreover, it can be noted that even if VirusTotal and OPSWAT share a certain common subset of antivirus scanning engines, the outcome of the conclusion might be varied. For instance, the antivirus scanner `BitDefender` has achieved the highest rates of positive detection (74.74%) from the OPSWAT positive reports while it performed at a much lower detection rate (10.02%) from VirusTotal. Similar case with `McAfee` antivirus scanner contributing for 88.96% of the positive reports in VirusTotal and 0.14% of the positive reports from OPSWAT.

Despite these varied performances of antivirus scanners, VirusTotal and OPSWAT reach the same or a consistent conclusion 95.79% of the time for a given malware sample as outlined in Section 6.1 and the redundancy of the reports from these services offer increased reliability and form a fail-safe feature for when reports cannot be retrieved with limited impact on the overall performance of the tool.

7.2 Source-code features results and analysis

The source code features that include API methods and string constants. API methods are presented as a frequency of how many times they are used in the source code of the application. This enables the user to trace which methods are used and how frequently. Figure 12 illustrates the frequency of API methods along with their corresponding API classes for a randomly picked APK sample.

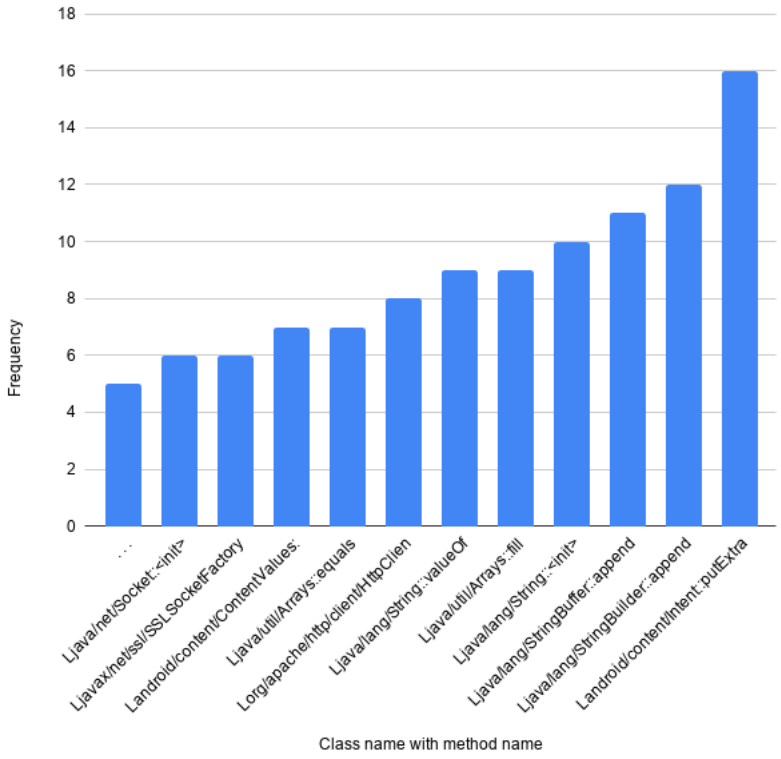


Figure 12: Frequency of API methods for a sample APK

It can be seen that this particular sample uses the `putExtra(...)` API method the most (16 times) and this API method belongs to the `android/content/intent` class. It uses the `append(...)` method from the class `StringBuilder` second most frequently at 12 times and so on. The API methods, particularly related to `android`, provide some vital insights when studying suspected malicious behavior of an Android application.

In the case of string literals, from the gathered data of most applications both from benign and malware samples exhibited some possible string obfuscation. Some of the string literals contained codes written in JavaScript while others included strings that are non Unicode based or an encoding such as base 64.

8 Conclusion

The Ultimate Feature Extractor tool provides a simple component based interface for aggregating sets of contextual and technical features on sets of Android applications. These sets of data provide insightful and applicable sets of information in the context of mobile security research. To answer the research question, gathering the contextual and technical features that have been examined in this paper has the effect of providing worthwhile sets of information that can be utilized in a variety of ways in different contexts. The sets of compiled data contribute to the process of identification, analysis and classification of Android applications on basis of security, privacy and other similar fields.

The tool is designed to incorporate commonly sought after features and it is made to facilitate the data gathering process in the mobile security research pipeline. To limit the burdens that can occur from data that consists of several features at once, the tool provides a configuration in which the user can prioritize which clusters of features they want to collect by enabling or disabling components.

The tool is also made with the priority of delivering data straight from the Android package file to the user as the tool's main goal is primary extracting and presenting data.

The tool has certain limitations and some features that have yet to be incorporated as part of the future work of the tool (Section 9). The amount of reports that can be gathered from virus scanning services have set quotas. As the tool integrates several scanning services simultaneously for reduced chances of missing reports, there can be some occurrences of missing reports. In the case, an access token is present that has no quota restrictions, then this limitation will be fully alleviated as this required token is parameterized under the configuration of the tool.

Overall, the tool is a worthwhile data gathering utility for works related to Android security. Users can utilize the gathered information for statistically based classification or clustering algorithms or otherwise for extensively investigating their own Android application of interest. It also incorporates other details to aid the research process by offering simple installation, configuration and execution of the tool with additional productive features such as progress tracking and error handling to improve the overall interaction with the tool.

The work related to gathering features evolves over time and has areas for exploration and growth. The tool certainly has areas for that can be improved and extended. Our work provides some basic model or a basis for an ever improving, iterative but yet extensive Ultimate Feature Extractor tool with the main goal of advancing/streamlining the research done on the security of mobile device applications.

9 Future work

There are a variety of recommendations to be made to enhance or improvement the performance or the robustness of the tool.

These improvements or works are outlined as follows.

- Support for parallelization or concurrent processing of tasks. As batch requests are presented for the tool, the given APKs can be handled in parallel manner with an approach that pays careful attention to safety when parallel processes manipulate the output files. The components can be made to be executed in parallel for a given set of tasks.
- Support for gathering permissions from Google Play as part of contextual data. This permissions list gathered from Google Play can be compared with the permissions from the manifest file to detect possible repacking of applications.
- Support for Dynamic run-time analysis component. There are sets of data that can only be acquired by running a given Android application. Researchers analyze certain run-time behaviors to identify activities that are obfuscated and hard to detect using static analysis.
- Support for an improved and a more robust way of detecting String obfuscation. Detecting the encryption of strings at a statistical significant rate is one of the most challenging tasks of information security research. The limited (probability based) string encryption detection that we present in our tool has areas to be iteratively improved or reworked for tasks in the future.

References

[1] Steven Arzt. Flowdroid static data flow tracker, Jan 2019.

[2] Anthon Desnos. Androguard, reverse engineering, malware and goodware analysis of android applications, Feb 2019.

[3] Android Developers. Documentation for app developers.

[4] Intellij developers. Fernflower, analytical decompiler for java, 2017.

[5] Shuaike Dong and et al. Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[6] Patrik Lantz. Droidbox, dynamic analysis of android apps, Aug 2014.

[7] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. Android malware clustering through malicious payload mining, September 2017.

[8] Rahman M, Carbunar B, and Chau DH. Search rank fraud and malware detection in google play. *IEEE Transactions on Knowledge and Data Engineering*, 29:1329–1342, June 2017.

[9] Alejandro Martín García, Raul Lara-Cabrera, and David Camacho. Android malware detection through hybrid features fusion and ensemble classifiers: The andropytool framework and the omnidroid dataset. *Information Fusion*, 52, 12 2018.

[10] Eugene Minibaev. Static dalvik vm bytecode instrumentation. 06 2017.

[11] Bob Pan. dex2jar, tools to work with android .dex and java .class files, Jun 2015.

[12] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 300–305, 2013.

[13] Benjamin Peterson. Pep 373 – python 2.7 release schedule, April 2014.

[14] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). *Future Generation Computer Systems*, 9:389–419, March 2016.

[15] Connor Tumbleson and Ryszard Wiśniewski. Apktool, a tool for reverse engineering 3rd party, closed, binary android apps, Nov 2019.

[16] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play?: a large-scale empirical study. *Proceedings of the 15th International Conference on Mining Software*, 29:231—242, May 2018.

[17] Wei Wang and et al. Constructing features for detecting android malicious applications: Issues, taxonomy and directions. *IEEE access 7*, 7(10):67602—67631, 2019.

[18] Xing Wang and et al. Characterizing android apps' behavior for effective detection of malapps at large scale. *Future Generation Computer Systems*, 75:30–45, 2017.