University of Groningen

Bachelor Thesis

Ultimate feature extractor for Android mobile applications - Technical Features

Author:
Haoran Xia *(s3470334)*

Supervisors:
Dr. Fadi Mohsen, PhD
Dr. Fatih Turkmen, PhD

July 18, 2020

Abstract

Android systems have long had the ability to install third party software through unofficial sources. This has resulted in these systems being a prime target for malicious intent. Current research has focused on extracting security features from Android applications and determining whether they are a safety threat or not using machine learning algorithms and other methods. However, researchers tend to use their own tools and data which leads to incomparable research results across different research. Thus we would like to propose an ultimate feature extractor that is able to combine different kind of features from different categories into one, with the possibility of adding and/or removing features as the user wishes. In this paper we propose the technical feature extraction part of the feature extractor. Such features includes information that can be obtained by analyzing files in an Android application.

Contents

1 Introduction

Android systems are a popular platform for mobile devices, currently dominating the mobile operating system market by a huge margin [1], it has thus become an equally large platform for malicious intent. Android systems, unlike iOS systems, allow users to install applications in the form of APKs through various kinds of sources other than the official Google Play Store, which has security measures in place against malicious apps. These alternative sources include Torrents, direct downloads, and third-party markets. However most unofficial sources do not possess such preventive measures. This means that Android application developers can easily include malware in their products and unknowing end-users could end up executing malicious code on their devices.

One common method to classify an application as benign or malicious is by extracting certain properties from the application and then classifying the app based on these properties. These properties are called features and are often used by researchers to identify maliciousness of an application. We will go more in-depth into this later on.

However, research groups tend to extract and parse features using their own tools. These tools are also often specialized for a particular subset of features so researchers that wish to use many features might have to acquire multiple tools. The fact that tools tend to be specialized and independently developed makes it often hard to compare results and combine data from such tools. Research produced using different tools can thus also be hard to compare. That is why we have developed an universal feature extractor that can extract various features most often used by researchers. Our aim is to provide researchers with an universal tool that helps them extract any desired features from Android applications. This removes the need to develop their own personalized extractor. This also allows researchers to compare results as the extracted features would follow the same formatting.

Thus we shall tackle the following research question:

1. **How do we develop an ultimate feature extractor?**

To answer this question we must look at several sub-questions that as a sum will answer the main question:

1. What features should we aim to extract?
2. Can we make use of existing features, tools, and/or frameworks to achieve our goal?

As a final note, the goal of this particular paper is to mainly focus on a subset of features implemented in the Ultimate Feature Extractor. We will be looking into what technical features, especially static technical features, are often used for Android security analysis and why those specific features were chosen. The contextual feature part and a subset of technifal features of the tool are implemented by my partner Yona Moreda. We designed several overlapping components together such as the application structure, and input and output formatting.

2 Background Information

Every Android application must follow a certain structure in the form of an APK. This means that a systematic method can be developed to extract information from an application. These bits of information are often called features and can be split into several categories: contextual features and technical features. Technical features can be split into two further categories namely static and dynamic features.

2.1 APK: Android application package

An APK is a package that contains the files required for an Android application. An APK usually contains the following files and directories:

1. **META-INF**: Directory created when signing the APK after compilation.
   (a) **MANIFEST.MF**: This manifest file contains information that is used by the Java run-time environment when loading the APK. Information such as where the Main class is resides here.
   (b) **CERT.SF**: This file contains a list of all the files and their SHA-1 digest.
   (c) **CERT.RSA**: This file contains the signed contents of the CERT.SF file.
2. **lib**: directory containing compiled code. This directory may contain multiple subfolders where compiled code exists for specific hardware architectures.
3. **res**: directory containing resources that are not compiled into resources.arsc
4. **resources.arsc**: file containing precompiled resources.
5. **assets**: This folder contains any media files. These files can be retrieved using the AssetManager class.
6. **DEX files**: The actual sourcecode of the application is compiled to DEX format and can be found in these files. The original sourcecode is usually Java (or another JVM interpreted language such as Kotlin).

7. **AndroidManifest.xml**: An additional manifest file that contains information about the application. Information such as package name, app components, permissions, and more.

In this project we mainly focus on the **AndroidManifest.XML** file and the **DEX files**. This is because the features that are most often extracted originate from these files.

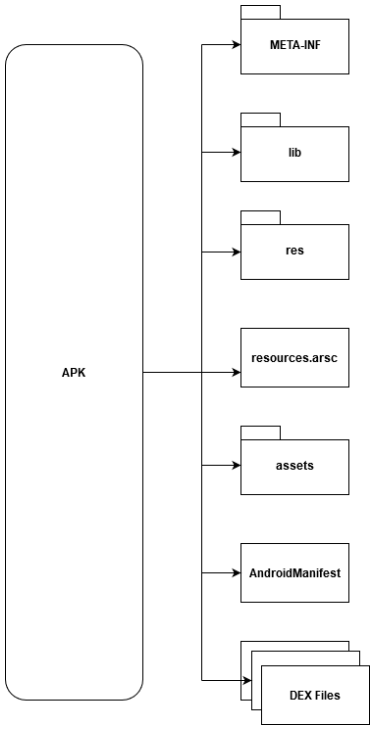


Figure 1: APK Structure overview

2.2 Contextual features

Contextual features are obtained without analyzing the APK itself. Information that can be found on the internet (or any other context) that describes the application such as developer name, application rating, and virus-scanner results belong to this category.

2.3 Technical features

Technical features are bits of information that can be extracted by analyzing the application. Obtaining such information requires unpacking an APK and analyzing the different files inside. Such an analysis method is called static analysis. Besides unpacking an APK it is also possible to execute an application and investigate its behaviour. Obtaining information from a running application is called dynamic analysis.

Static features

Static features belong in the technical feature category and can be obtained without executing the application. This makes it an efficient feature to extract due there being no overhead of running an application. Examples of static features would be manifest file information, source code patterns, compiled bytecode patterns, more or less anything inside the APK file. Especially the manifest file is an efficient source for feature extraction. Since the file contains application information in XML format, it can easily be parsed and useful information such as permissions can be extracted quickly.

Dynamic features

Dynamic features are often harder to extract. Such features often require some environment to run the application in and the generation of user input before behaviour can be observed [2]. When the application emits behaviour the information is logged and further analyzed afterwards. This information may include network data, file read/write information, and more.

3 Utilized tools

There already exist many tools out there that can aid in analyzing an APK file. Tools that decrypt an APK, extract the different files in there. Tools that allow us to decompile DEX files into Java sourcecode, and more.
For our specific project we utilize a framework called Androguard for most technical feature extraction. We have also added a second option for analyzing and decompiling DEX files using dex2jar and the fernflower decompiler.

3.1 Work environment

The development and testing of the tool has been done on both Windows (10) and Linux (Ubuntu) operating systems. As for the tool itself, Python version 3.8 has been used during the development.

3.2 Datasets

The datasets used were all provided by professor F.D Mohsen. The datasets were split into benign and malware APK samples and we extracted features with those categories in mind.

3.3 Androguard

Androguard [3] is a Python based framework that allows us to inspect and investigate APK files. It contains functionality for almost everything that we wish to accomplish. It is a framework that has been used by many applications that perform some form of Android security related task [3]. In short, Androguard allows us to call functions that parse the APK file. These functions return objects containing a plethora of methods and fields that aid us in our feature extraction.
For our specific use we mainly use Androguard for manifest file and source code analysis.

Manifest file functionality

The Androguard function **AnalyzeAPK()** returns an Androguard.APK object containing functionality that allows us to extract meta-data from the APK file. We mainly use this object to extract data from the AndroidManifest.XML file. In our implementation, the APK object is used to extract features such as the package name, permissions, used hardware features,

used software features, and many more useful bits of information. A comprehensive list is available in the implementation section.

Source code functionality

Androguard also allows us to peek into the decompiled sourcecode and compiled bytecode of the APK. Such information resides in the DEX files. We construct the nessecary Androguard objects and use the default decompiler (DAD) that comes with the Androguard framework for any Java sourcecode related features.
When we do sourcecode analysis we construct two Androguard objects, to be specific the following two:

1. **d - object**: This is an array of Androguard.DalvikVMFormat objects. A DalvikVMFormat object contains information about one DEX file in the APK. If there are multiple DEX files then we must associate each DEX file with a DalvikVMFormat object. Regardless of the fact whether one there is one or more DEX files, the DalvikVMFormat objects are put into an array. The DalvikVMFormat object contains a plethora of functionality that allows us to do things such as getting the classes, classnames, methods, sourcecode, and more.

2. **dx - object**: An Androguard.Analysis object. This object contains information about multiple DEX files. If the APK has multiple DEX files then we can construct a DalvikVMFormat for each DEX file and from there we can link them to the Analysis object. The main reason to use the Analysis object for multiple DEX files is that it contains special functionality for extracting features across multiple DEX files.

3.4 Dex2jar

Dex2jar [4] is a lightweight tool that allows us to convert DEX files into Jar files containing Java class files. We use this tool as an intermediate step to go from DEX files to Java source code files.

3.5 Fernflower

Fernflower [5] is a decompiler for Jar files containing Java class files. It is the build-in decompiler for the Intellij IDE and is thus also developed by Jetbrains. After feeding the tool with a Jar file containing Java class files it generates a Jar file with Java sourcecode files inside. Using simple techniques we can read these files as a string and use regex expressions to search the sourcecode for anything we wish for.

4 Implementation

In this section an overview is given about the layout of the tool. We will talk about the different features and functionality that the tool provides and design decisions.

4.1 Introduction

The main idea of the project was to develop an extendable tool that can extract different features from an APK. We chose Python for this purpose. Python is a high-level language that is readable and easy to work with. Besides this, the fact that Androguard is a Python framework was also a big motivation for choosing this language. Androguard can be seen as the backbone for our project. The existence of this framework made it a lot easier to develop our feature extractor, and as can be seen from the implementation, most of the features that we extract rely on Androguard provided functionality.

4.2 Tool overview

Our tool is a simple Python 3 program that can be run on the command shell. The program accepts a folder of APKs or a single APK. Included with the program is a settings.ini file that gives the user full freedom of what features to enable and/or disable. Such a file also allows future developers to easily add their own modules and optionality for enabling them.

4.3 Initialization and use

Python comes with many frameworks and tools out there that helps us in managing a project. For this reason we use a tool called Pipenv. Our project uses many libraries, Pipenv allows us to manage all of this properly. Before using the tool, it is recommended to create a Pipenv environment with the command:

```
pipenv shell
```

After this is done all the required packages can be installed with the command:

```
pipenv install
```

The resulting state is that an environment for this specific project is created with all the required packages residing in that environment. Note that outside of the environment the packages are not installed unless they are installed globally.

To use the application we have to activate the environment first by using the command:

```
pipenv shell
```

Once the environment has been started we can start extracting features from a single APK or a batch of APKs using the following commands:

1. **Single APK execution mode:** `python main.py -sAPK <path-to-apk>`
2. **Batch APK execution mode:** `python main.py -s <path-to-apks-folder>`

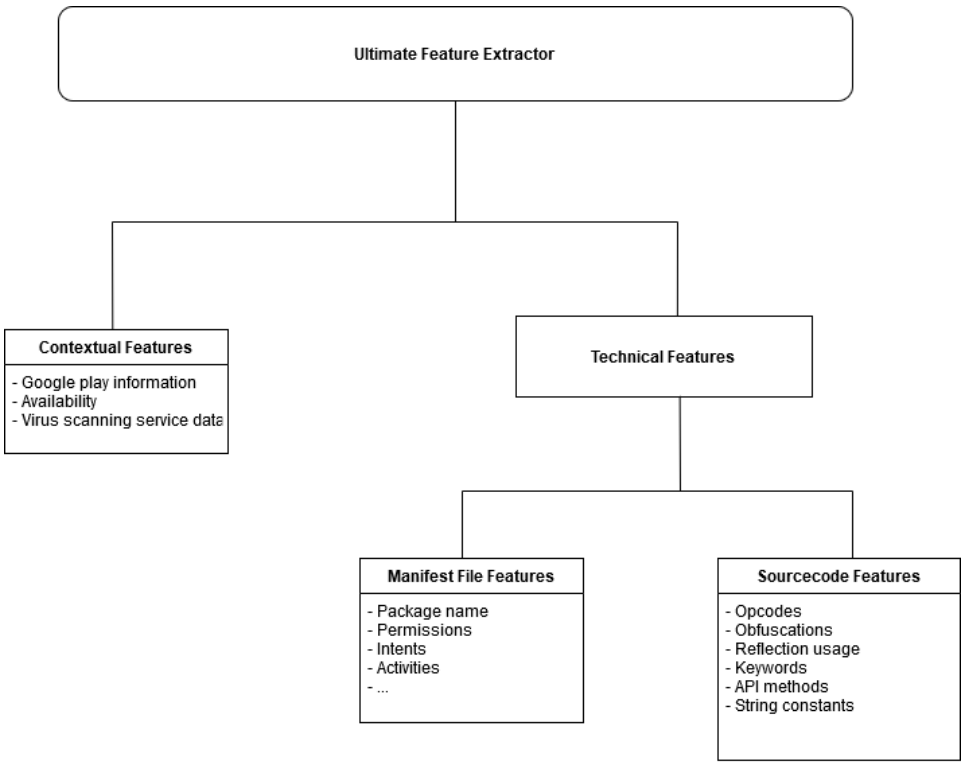


Figure 2: Overview of the Ultimate Feature Extraction Tool

4.4 Features

There exist much research work in the field of Android feature extraction. This means that we have to be careful as to pick what features we will be extracting and what features we will be omitting. First of all, our focus is on the static technical side of feature extraction. This means that we can already omit a big list of features. We have researched existing papers and found one particular source that investigated what features other researchers tend to use for their work [6]. Using this work as a general guideline, we have thus investigated research work that performed actual feature extraction and based our tool on what features were most commonly obtained. Some of the features we extract are: API calls, intents, App components, Operation Code (Opcodes), String constants, and more. Note that my partner worked on contextual features and a subset of technical features such as API calls and String constants.

4.5 Manifest file features

Manifest file features belong to the category of static technical features. The manifest file contains information that describes the application and is in an easy to parse format (XML). This means that we can quickly extract information from it. One thing to note that this file is initially encoded however Androguard automatically decodes this file once you construct the appropriate object.

We extract manifest file features using the Androguard framework. The Androguard framework allows us to construct an **APK** object. By calling the constructor for this class in the **androguard.core.bytecodes.apk** module. We then pass the **APK** object to a function that handles the extraction of manifest file features. All the feature extraction is done using functionality that the **APK** object provides.

An example of Androguard functions that we use can be found below:

```
1 apk.get_package()     //Gets the package name
2 apk.get_permissions() //Gets the permissions
3 apk.get_features()    //Gets the hardware/software features used
4 ....
```

Extracted features

We extract the following information from the manifest file:

1. **Package name**: The package name identifies the application.

2. **Permissons**: When an application wishes to access sensitive user data or system features it needs to ask for permissions first. These permissions are to be described in the AndroidManifest.XML file and by extracting them we can get an general impression about the APK in question.
   One thing to note is that an application can define their own permissions. This means that if a third-party application wishes to use another third-party application's features, it has to ask for permissions from that application first. However, a comprehensive list of such permissions can not be complete so for our tool we allow the user to provide their own list of permissions in a txt format. By default we use the well defined Android system permissions [7]

3. **Used features**: Android applications can make use of features that are present on the device. Features are split into hardware features such as the camera, microphone, and software features such as webview (the ability to display content from the internet in the app), and many other things. The features that we extract are the ones defined in the Android documentation [8]. We also give the user the option to provide their own list of features to extract.

4. **Activities**: Activities are the entry point for interaction with the user. When an app is invoked by the user or another app, an activity is called to determine what to happen next. The features we collect are the names of the activities present in the application.

5. **Services**: Services are a component in Android applications that perform long-running operations in the background. For example, a music application may be playing music in the background while the user is using some other application.

6. **Receivers**: Receivers are places that receive messages from various kinds of sources. For example when the phone enters airplane mode, a message is sent to applications to let them know what is happening.

7. **(Content) Providers**: This component supplies data from application to application upon request. Content providers allows you to centralize data in one area and have different applications request or modify data from it. It can be seen as a database for data.

8. **Intent filters**: Intents are objects which request actions from another app component. Intent filters are the places that receive the corresponding intents. If an intent filter matches an intent then that component is started to handle the received intent.

9. **Libraries**: If the application uses some specific shared library that the application must be linked against then it should be specified using these tags.

Manifest feature extraction motivation

As mentioned before, manifest file are cheap and easy to extract. The Androguard framework makes it especially easy to extract as it comes with a plethora of functions to extract specific data. Manifest file are often used in combination with classification algorithms where the extracted features are used to train such an algorithm so that it is able to classify unlabeled data into benign or malicious [9] [10] [11].

4.6 Sourcecode features

Sourcecode features belong to the static technical features category. We extract sourcecode features using the Androguard framework. We initialize the required objects, **(d - DalvikVMFormat, dx - Analysis)** objects, and then pass them on to our functions that handle sourcecode feature extraction.
In our tool we have opted to extract three main features. These features are:

1. **Opcodes**: We can obtain Dalvik opcodes by analyzing the bytecode using Androguard. This can give us an insight into application behaviour on a low level. Opcodes have been used as a feature in various research work [12] [13] [14]. The opcodes that we extract are consistent with the Android defined opcodes that can be found in the documentation [15].

2. **Identifier obfuscation**: A common way of obfuscation is by replacing identifier names (variables, classnames, etc..) with meaningless

gibberish such as 'a', 'aa', 'aaa'. [16]. Popular obfuscation tools such as Proguard, Allatori, and many more use this technique.

3. **Java reflection usage**: Reflection is a Java specific feature that allows the developer to hide program behaviour. With reflection the developer can call functionality from other classes in a roundabout way which can be hard to detect by many analysis tools [16]. Thus reflection might be an indication of malicious intent.

We extract identifier obfuscation and Java reflection usage features due to the fact that we were requested to extract features related to code obfuscation. These two features may give an indication of whether obfuscation exists within the apk or not.

Opcodes implementation

Opcodes are one of the features that can be extracted using Androguard. We can use the d (**DalvikVMFormat**) object to iterate over the classes of the DEX file. Then we iterate over the methods for each class, followed by an iteration over **DalvikCode** objects, and finally we can obtain a list of instructions or as we call it: Opcodes. Opcodes are in general a popular feature for extraction and have been used before by other researchers.

Identifier obfuscation implementation

To get identifiers we again use Androguard functionality. Androguard provides functions for us to loop over the classes, fields, and methods, from which we can obtain the associated identifiers. After obtaining these we perform several checks to determine whether it might be a possible obfuscation or not. These checks include checking the identifier for length and checking whether they are in valid ASCII notation. Note that our goal is not to determine whether an application is one hundred percent obfuscated, but rather to give an estimation whether there are indications of obfuscation or not.

Java reflection usage implementation

Unfortunately there are no Androguard build-in functions for finding reflection usage. This means our approach was to obtain the decompiled sourcecode and then match the sourcecode with specific regex patterns. DAD decompiled sourcecode has reflection calls in the format of the following regex expression:

```
reflect.([a-zA-Z]+)
```

So once we obtain the sourcecode as a string we use build-in python functionality to search the string for matches using our regex expression.

The following code snippet shows an example of how reflection calls look like when the sourcecode is decompiled. Note with the specified regex expression in the section above we extract statements such as:

```
reflect.Array
```

This is due to the fact that we only match for the corresponding reflection library. We do not care what functionality is used from that library.

```
protected void zzc(Objcet p4, com.google.android.gms.internal.zzbum
    p5)
{
    int v1 = reflect.Array.getLength(p4);
    int v0 = 0;
    while (v0 < v1) {
        Object v2 = reflect.Array.get(p4, v0);
        if (v2 != null) {
            this.zzb(v2, p5);
        }
        v0++;
    }
}
```

Alternative method

The only discussed methods so far for sourcecode feature extraction have been related to the Androguard framework. Although Androguard provides very good functionality for this cause, using the decompiled sourcecode directly is not that pleasant compared to some alternative options. Decompiled sourcecode using DAD, the Androguard decompiler, is often not that close to the true sourcecode and often lacks various bits of information such as import statements. For this reason we have also implemented an alternative method for decompiling sourcecode and inspecting it. This method makes use of the Fernflower decompiler [5] developed by JetBrains.

Using Fernflower

Fernflower is a tool that does not work directly with DEX files. We first have to convert the DEX files to a Jar file containing Java class files. This is done by utilizing another tool called dex2jar [4]. In our program we first input the APK into the dex2jar program which produces a jar file as output. This file is then given as input to Fernflower which as a result also produces a jar file. However this jar file contains java sourcecode files that we are looking for.

The implementation of using dex2jar and Fernflower in our tool is done by using system calls defined in the `python sys` library.

Extracting features

Fernflower outputs a jar file containing java source code. We read these files and treat them as a string so that we can again use regex pattern matching techniques on them. Fernflower only produces decompiled sourcecode. This means that some of the features we are extracting using Androguard are not applicable. We opted to go for just regex pattern matching to extract reflection usage and import statements. This is because Androguard constructs objects using the decompiled sourcecode as data. These objects contain functions that parse the sourcecode and give us an easy method for getting information such as variable names and more.

The regex pattern we use to extract import statement is:

```
import (.*?);
```

The regex pattern for extracting reflection statement is:

```
java.lang.reflect.*;
```

We also count the number of failed decompilations that occur when using fernflower. Whenever fernflower fails to decompile a certain class or method it replaces the actual sourcecode with a piece of text that can be matched using the following regex expression:

```
// FF: Couldn't be decompiled
```

The following code snippet shows an example of how import statements look when decompiling code using Fernflower.

```
1 import java.lang.annotation.Retention;
2
3 import java.lang.annotaton.RetentionPolciy;
4
5 import java.lang.reflect.Constructor;
6
7 import java.lang.ArrayList;
```

The following code snippet shows an example of methods that failed to decompile using Fernflower.

```
1 public static Bundle a(Notification param0){
2     // $FF: Couldn't be decompiled
3 }
```

4.7 Miscellaneous features

Apart from the feature extraction features our tool also comes with several quality of life features that makes using the application more convenient. We have added support for logging, progress tracking, and optionality.

Logging

Logging is enabled by default and displays useful messages to notify the user about the progress and status of the tool. Information such as decompilation duration, analysis duration, errors during analysis and more are logged.

An example of a log entry may look as follows:

```
INFO:root:Processing apk: <apk_name> || file: <file_path>
INFO:root:Running manifest
INFO:root:Running sourcecode
INFO:androguard.analysis:End of creating cross references (XREF)
INFO:androguard.analysis:run time: <time>
INFO:root:Time spent on opcodes: <time>
INFO:root:Time spent on obfuscation: <time>
INFO:root:Time spent on keyword usage: <time>
INFO:root:Apk: <time> || Time spent on analysis: <time>
INFO:root:Updating progresstracker file
INFO:root:Total time spent on this apk: <time>
```

Progress tracking

Our tool allows the tracking of the current progress when a list of apks is given as input. We log the file name (not package name) and save it in a txt format. Whenever we analyze a new apk we check the file against this list to determine whether it has already been processed or not. Note that the implementation of the progress tracking feature is quite rudimentary. The algorithm may be optimized by taking advantage of the ordering of apk files for example. As of now we loop over the whole txt file to determine whether the apk file has already been processed or not. This means the algorithm that does progress tracking has time complexity of $O(n)$ where n is the number of entries in the txt file.

The following snippet shows an example of how files are saved in the txt file:

```
009dad2e09e5d50e3b5bc3e8bbff5dee.apk
07d8b00632e70367f40cd402b1f71dd7.apk
08e2cc8d74b38136a2d99b585b09c278.apk
5dc0d5178be9b4ca9f0a8e979223f02b.apk
5dcff0234b554fdfbf39d82d735924ae.apk
5def5791587edd79923693d590bf6471.apk
5e415996c602db10b78ece34491b0b38.apk
5eaf4066f4386a20e6db77c4e80513c3.apk
5eb792298f156c4a9ee4ee532f1a2170.apk
5f187d859de2012958c1c013e2ebcd37.apk
5f216304dd220b70189e914219f92c1f.apk
5f5c9e81ce53b43fb6612fc00db9abda.apk
5f666c7ca54bb6e535aaeeffbbaac27a.apk
5f6b28e8f331652f761dd6565e06f900.apk
5f975bfe4836b1341b1fe778dfa25f06.apk
5f9ef3c03cf301ee49b9f594b73ee6e7.apk
5fa2047e2ddc3b1faa5b48c3d0638c24.apk
5fbd3e307150e4b3eaa0ad8f8d4612b5.apk
5fe71132d1a8ea013474a929a4347309.apk
5ffa85f91a4dcdb00242c4fc6bb25fef.apk
```

Optionality

The tool is designed with modularity in mind. Each feature in the tool, ranging from extraction features to quality of life features, can be enabled or disabled at will to allow for flexible usage of the application. We implemented this by making use of the **INI** file format.

The following snippet shows an example of part of our **Settings.INI** file

```
[Settings]
Contextual = no
Manifest = no
Sourcecode = no
Fernflower = yes
Performancelogging = no


[Contextual_Settings]
opswat = yes
app_store = yes
virus_total = yes
hybrid_analysis = yes
```

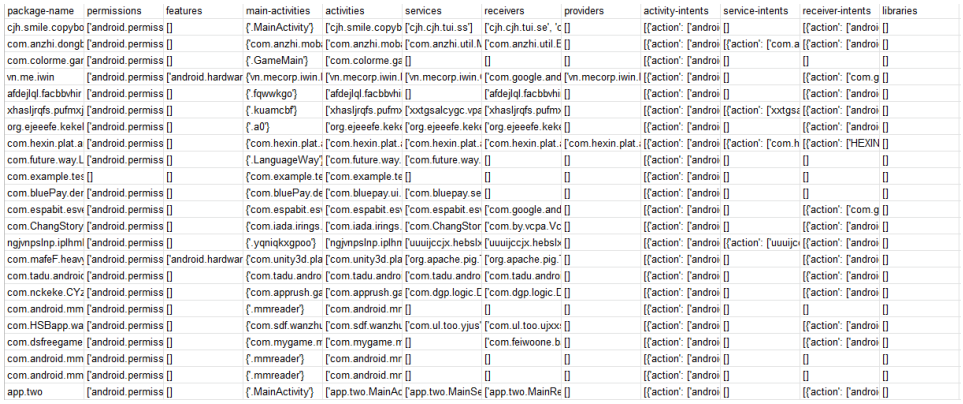
```
virus_total_enable_file_upload = yes
opswat_api_key = <api_key>
virus_total_api_key = <api_key>
hybrid_analysis_api_key = <api_key>


[Sourcecode_Settings]
Opcodes = yes
Obfuscation = yes
Keywordusage = yes
Kotlin = yes
Reflection = yes
Commonkeywords = yes
StringConstants = yes
APIMethods = yes
```

4.8 Output formatting

To display the extracted properly we must format the obtained data first. The data that our csv functions expect require the package name (not file name) as the key (unique identifier). As long as we format the output properly to accommodate this expectation the functionality that handles csv writing can be properly used without errors.

CSV Format

By default we store the extracted features in csv files. CSV files provide an easy and quick way to represent all the data we collect. The extracted features are stored in rows with the package-name as the key and the extracted features in the columns. CSV allows us to quickly and easily extract data per column which gives us the desired features without much of a hassle.



package-name	permissions	features	main-activities	activities	services	receivers	providers	activity-intents	service-intents	receiver-intents	libraries
cjh.smile.copybo	['android.permiss	[]	{'.MainActivity'}	['cjh.smile.copyb	['cjh.cjh.tui.ss']	['cjh.cjh.tui.se', 'c	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.anzhi.dongt	['android.permiss	[]	{'com.anzhi.mob	['com.anzhi.mob	['com.anzhi.util.A	['com.anzhi.util.E	[]	[{'action': ['androi	[{'action': ['com.a	[{'action': ['androi	[]
com.colorme.gar	['android.permiss	[]	{'.GameMain'}	['com.colorme.ga	[]	[]	[]	[{'action': ['androi	[]	[]	[]
vn.me.iwin	['android.permiss	['android.hardwar	{'vn.mecorp.iwin.I	['vn.mecorp.iwin.I	['vn.mecorp.iwin.(	['com.google.and	['vn.mecorp.iwin.I	[{'action': ['androi	[]	[{'action': ['com.g	[]
afdejlql.facbbvhir	['android.permiss	[]	{'.fqwrwkgo'}	['afdejlql.facbbvhi	[]	['afdejlql.facbbvhi	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
xhasljrqfs.pufmxj	['android.permiss	[]	{'.kuamcbf'}	['xhasljrqfs.pufm>	['xxtgsalcygc.vpa	['xhasljrqfs.pufm>	[]	[{'action': ['androi	[{'action': ['xxtgsa	[{'action': ['androi	[]
org.ejeeefe.kekel	['android.permiss	[]	{'.a0'}	['org.ejeeefe.kek	['org.ejeeefe.kek	['org.ejeeefe.kek	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.hexin.plat.a	['android.permiss	[]	{'com.hexin.plat.a	['com.hexin.plat.a	['com.hexin.plat.a	['com.hexin.plat.a	['com.hexin.plat.a	[{'action': ['androi	[{'action': ['com.h	[{'action': ['HEXIN	[]
com.future.way.L	['android.permiss	[]	{'.LanguageWay'}	['com.future.way.	['com.future.way.	[]	[]	[{'action': ['androi	[]	[]	[]
com.example.tes	[]	[]	{'com.example.te	['com.example.te	[]	[]	[]	[{'action': ['androi	[]	[]	[]
com.bluePay.der	['android.permiss	[]	{'com.bluePay.de	['com.bluepay.ui.	['com.bluepay.se	[]	[]	[{'action': ['androi	[]	[]	[]
com.espabit.esv	['android.permiss	[]	{'com.espabit.esv	['com.espabit.esv	['com.espabit.esv	['com.google.and	[]	[{'action': ['androi	[]	[{'action': ['com.g	[]
com.ChangStory	['android.permiss	[]	{'com.iada.irings.	['com.iada.irings.	['com.ChangStor	['com.by.vcpa.Vc	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
ngjvnpslnp.iplhml	['android.permiss	[]	{'.yqniqkxgpoo'}	['ngjvnpslnp.iplhn	['uuuijccjx.hebsl>	['uuuijccjx.hebsl>	[]	[{'action': ['androi	[{'action': ['uuuijc	[{'action': ['androi	[]
com.mafeF.heav	['android.permiss	['android.hardwar	{'com.unity3d.pla	['com.unity3d.pla	['org.apache.pig.'	['org.apache.pig.'	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.tadu.androic	['android.permiss	[]	{'com.tadu.andro	['com.tadu.andro	['com.tadu.andro	['com.tadu.andro	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.nckeke.CY2	['android.permiss	[]	{'com.apprush.ga	['com.apprush.ga	['com.dgp.logic.D	['com.dgp.logic.D	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.android.mm	['android.permiss	[]	{'.mmreader'}	['com.android.mr	[]	[]	[]	[{'action': ['androi	[]	[]	[]
com.HSBapp.wa	['android.permiss	[]	{'com.sdf.wanzhu	['com.sdf.wanzhu	['com.ul.too.yjus'	['com.ul.too.ujxx>	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.dsfreegame	['android.permiss	[]	{'com.mygame.m	['com.mygame.m	[]	['com.feiwoone.b	[]	[{'action': ['androi	[]	[{'action': ['androi	[]
com.android.mm	['android.permiss	[]	{'.mmreader'}	['com.android.mr	[]	[]	[]	[{'action': ['androi	[]	[]	[]
com.android.mm	['android.permiss	[]	{'.mmreader'}	['com.android.mr	[]	[]	[]	[{'action': ['androi	[]	[]	[]
app.two	['android.permiss	[]	{'.MainActivity'}	['app.two.MainAc	['app.two.MainSe	['app.two.MainRe	[]	[{'action': ['androi	[]	[{'action': ['androi	[]

Figure 3: Manifest file features csv output. Each column is an array of features found. If the feature is not found then an empty array is produced.

JSON Format

We also have the option to produce output in JSON format. This output is identified per object where each object contains all the obtained features for one specific APK. In contrast to CSV format where we can address data per column, in JSON format we must identify data per object and then retrieve the desired data for a specific feature if it exists. Note however that JSON formatting tends to produce big files.

5 Results

We have already seen an example of what manifest file features can be used for. In general, the same purposes extend to the other features such as contextual features, sourcecode features, (and possibly dyanmic features). In this section we would like to expand on these purposes and give examples of what can be possible.

To test the tool we have ran the application using the datasets mentioned in the **Utilized tools** section. We have extracted features from both benign and malicious apks. Due to the fact that my contribution to this project only extends to technical features, the features that will be presented in this section will thus also belong in the technical category. This means that the features that have been extracted are the manifest file features, sourcecode features using Androguard (opcodes, identifier obfuscations, reflection usage), and Fernflower decompiled sourcecode (Import statements, failed decompilation counts, reflection usage). All this data can be analyzed in different ways. We shall take a look at two of them namely in the form of Statistics, and in the form of Artificial Intelligence.

5.1 Statistics

One simple method of analysis is producing statistics from the obtained data and then analysing the obtained statistics [17]. The datasets containing apks were originally split into benign and malicious sets. Using this fact we can obtain statistics from the extracted features such as common opcodes, popular permissions, popular used features, and more for both benign and malicious apks. We can then make comparisons and draw conclusions from these statistics and thus have a relatively quick and easy way for gaining insight into tendencies of benign and malicious applications.

Below you can find some statistics about benign and malicious apks. These statistics were generated from features extracted from a subset of 1000 benign apks and 1000 malicious apks.

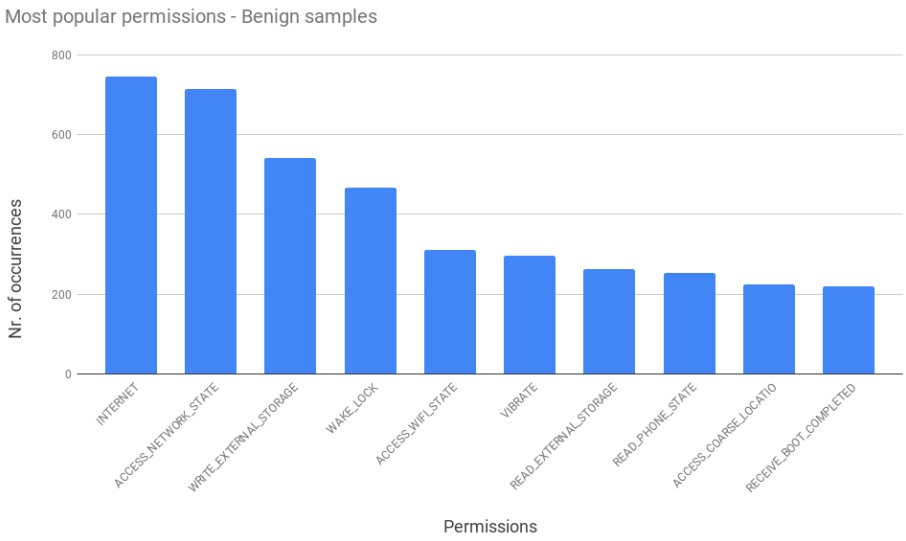


Figure 4: Statistics example 1 - Benign apks

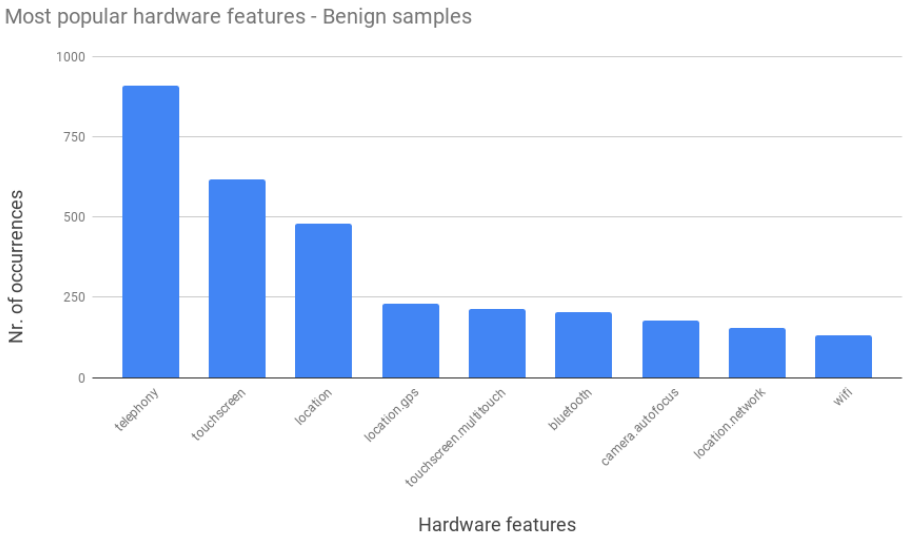


Figure 5: Statistics example 2 - Benign apks

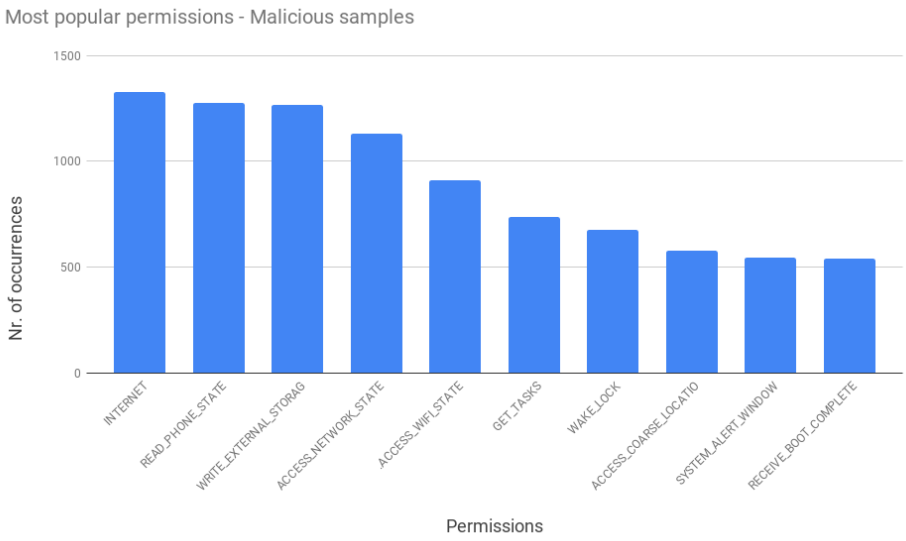


Figure 6: Statistics example 3 - Malicious apks

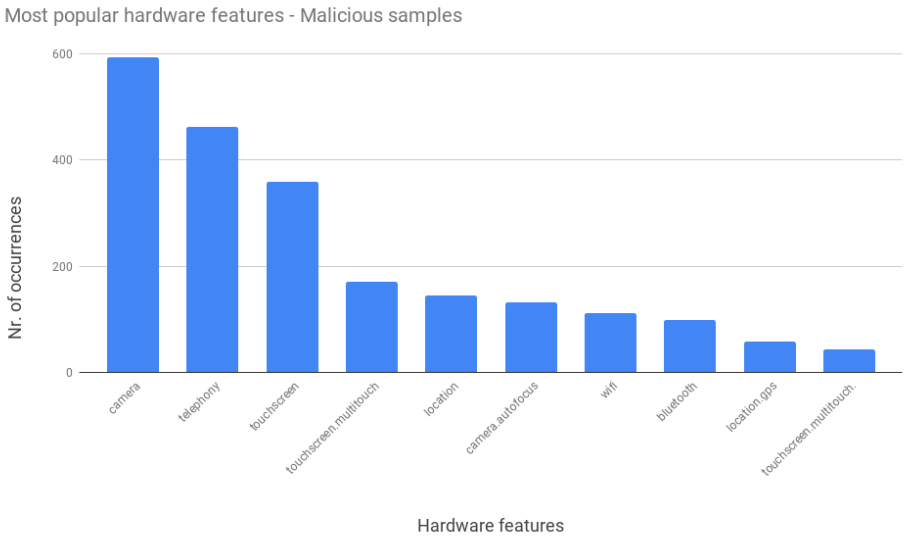


Figure 7: Statistics example 4 - Malicious apks

5.2 Artificial Intelligence

The data obtained from the tool can also be used as features in Artificial Intelligence algorithms. Artificial Intelligence have been employed before in the hopes of being able to accurately classify APKs into being benign or malicious.
Classification algorithms such as (Learning) Vector Quantization or K-nearest neighbour algorithm can classify applications into benign or malicious based on how thorough the algorithms have been trained using the obtained feature data. Another Artificial Intelligence technique that can be applied are Neural Networks. In the same way as the other classification algorithms, data obtained from the tool can be used to train the algorithms and then the algorithms will be able to classify APKs from a new dataset into malicious or benign by themselves.

AI examples

Some examples of the use of Artificial Intelligence algorithms with extracted features as feature data:

1. A paper by Xiangyu-Ju [9] who uses information from permissions and packages as features in classification algorithms/methods such as Bayes, KNN, and more.

2. Drebin [18] is a tool that attempts to classify an application as malicious by applying machine learning algorithms on extracted features.

3. Another paper by Ravi Kiran Varma P et al. [19] use extracted permissions as features for several machine learning algorithms. They then determine which algorithm performed the best on their dataset.

5.3 Other methods

Other methods which can not be easily classified into a category for analyzing extracted features also exist. For example, Ryo Sato et al. [20] proposes an alternative method for malware detection using the manifest file. In their work they compare the extracted manifest features with a predefined keyword list and calculate a malignancy score based on the comparisons. This score then aims to classify an application as benign or malicious.

6 Performance

Performance is also an important factor when analyzing large sets of data. In this section we are going to take a look at the performance of the manifest features and the sourcecode features with the DAD decompiler and Fernflower decompiler.

6.1 Manifest file features

Manifest file feature extraction is in general very quick. The main reasons for this is that it is just a XML file where we extract the required data. We collected performance data of this feature for 1584 apks. These apks were all labeled as benign and obfuscated.
We can observe that in general the feature extraction is less than one or two seconds. The extraction time also seems to increase with file size however it is not very drastic since almost all apks finish within two seconds regardless of size.
We can also spot some outliers. Such outliers may exist due to obfuscated manifest files for example.

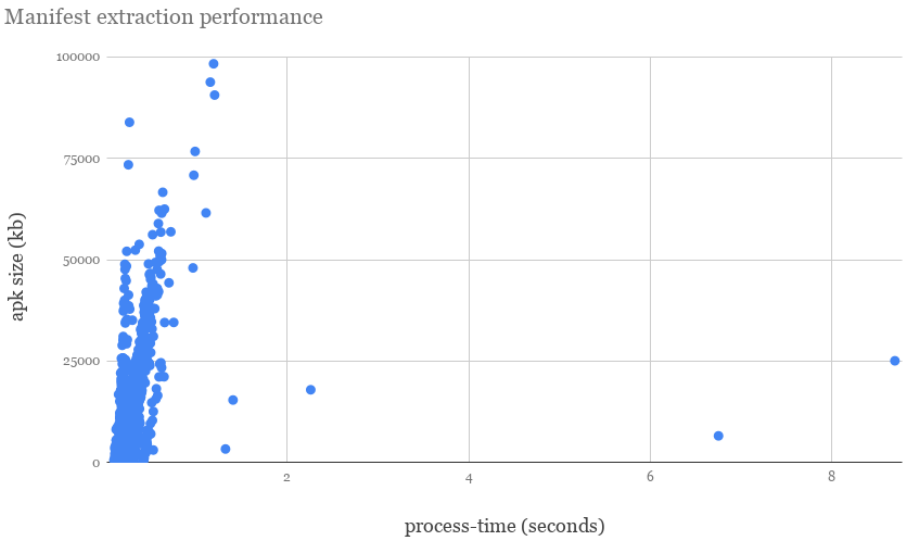


Figure 8: Scatterplot of manifest feature extraction performance

6.2 Sourcecode features

We can extract sourcecode features using either DAD or Fernflower. In this section we compare the two in terms of decompilation time. We collected this data using 182 malicious apks.

The performance of the DAD decompiler seems to follow a linear trend where process-time (decompile time) increases with apk-size.
Fernflower performance on the other hand seems to not follow a linear trend of decompile time versus apk-size. We can also observe that in general Fernflower takes much longer than decompiling code using DAD. Fernflower also contains some extreme outliers that take up to many hours to decompile. We have noticed from experience that heavily obfuscation APK files may be very troublesome for the Fernflower decompiler. This could be a reason for the (extreme) outliers.

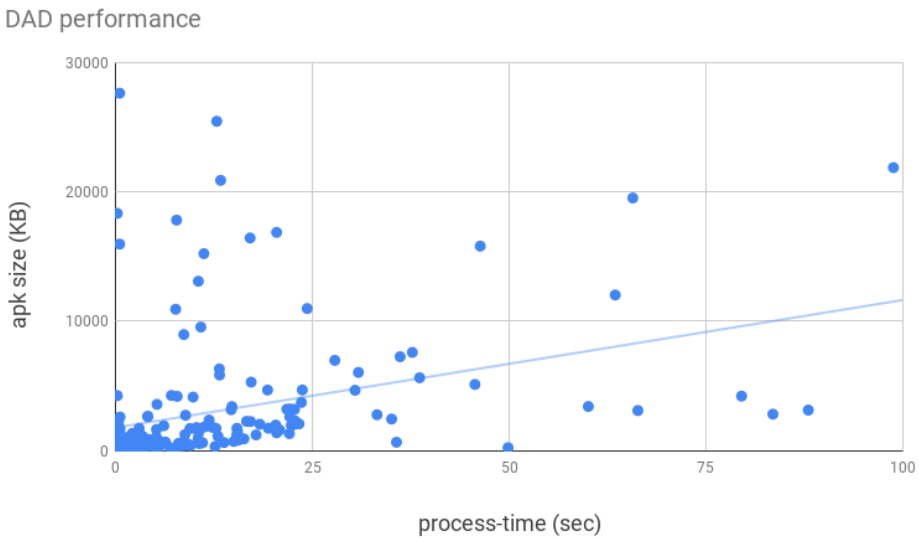


Figure 9: DAD decompiler performance.

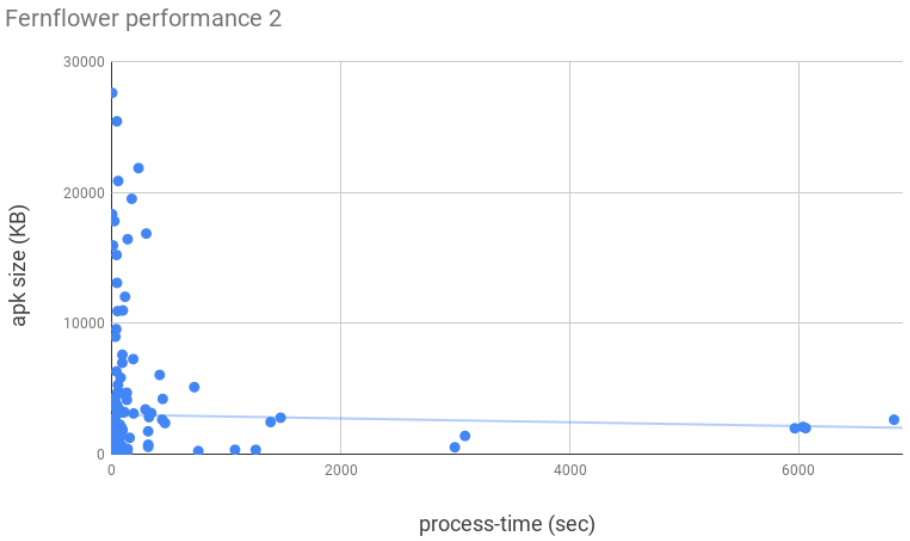


Figure 10: Fernflower decompiler performance.

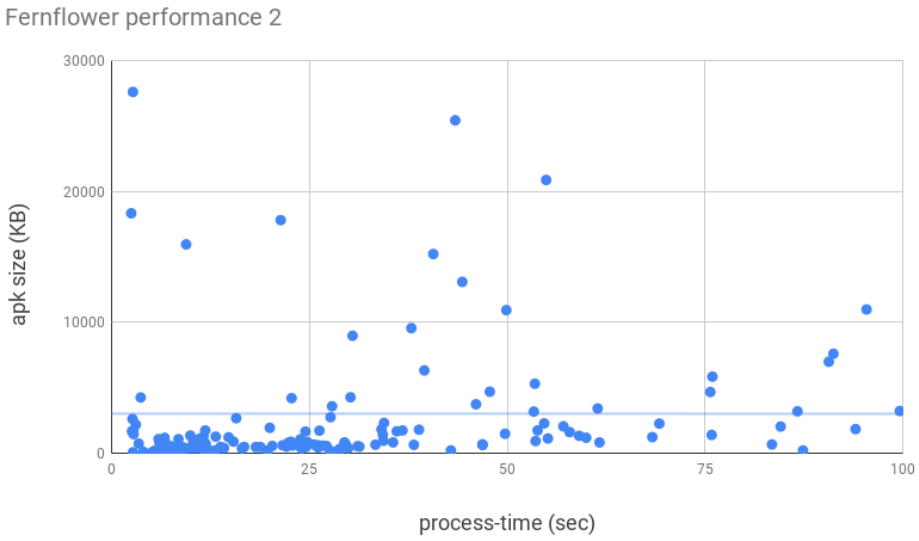


Figure 11: Fernflower decompiler performance zoomed in.

7 Future work

As of the writing of this paper the only features that have been employed are contextual feature extraction and static technical feature extraction. This means that we have not touched upon dynamic feature extraction yet which is the other big category of technical features that can be extracted from APKs. Aside from missing dynamic feature extraction, the currently implemented methods for contextual and static features can also be expanded on and alternative methods can possibly be found that achieve the same result.

7.1 Dynamic feature extraction

As mentioned before, dynamic feature extraction is a missing part of the tool. We have only developed part of the ultimate feature extraction tool by implementing contextual and static feature extraction. Dynamic feature extraction in itself is a different beast to tackle and requires its own research before being able to add it to the main tool. However once such a module has been developed it can easily be connected to the main program due to the fact that we structured the application with modularity in mind. Simply using system calls, or adding a call to the program that does dynamic feature extraction to the main program should accomplish the extension.

7.2 Alternative feature extraction methods

There exist other ways to analyze APKs and extract information outside of the used methods (Androguard, Fernflower). For example a tool such as `Apktool` [21] can be used instead of Androguard to decode APKs into some usable format. Furthermore the `AndroidManifest.XML` file can also be parsed manually using XML parsing libraries instead of making use of Androguard. In conclusion, many alternative tools exist for the same purpose and these tools may be added to the ultimate feature extractor at a later stage.

Outside of looking at alternatives we can also extend the ultimate feature extractor by implementing more features to extract. As mentioned before, the current tool mainly extracts the most popular features according to current research. However, an ultimate feature extractor is difficult to finalize because of the fact that many different kinds of features exist. Depending on the definition of "ultimate" we can keep adding features iteratively as the new research discovers more useful features to extract.

7.3 Performance enhancements

One final thing to note is that as of the writing of this paper the different modules and features of the ultimate feature extractor is executed sequentially. This leaves room for improvement as several modules/components do not necessarily rely on each other and can thus be executed in parallel. Some examples for this are the contextual, DAD sourcecode feature extraction, and Fernflower sourcecode feature extraction modules. These components do not depend on eachother when they are retrieving data so they could be ran in parallel.
One final thing to note is that the only conflicting part of these modules could be the writing to output section of the code. However, each module tends to write to their own dedicated output files. This means that as long as future extensions keep this in mind then conflicts or race conditions due to parallelism can be avoided.

8 Conclusion

When looking at feature extractor tools out there we can notice that most of them only extract data from a certain subset of features, are catered to a specific use, or are very outdated and ill-maintained. We have thus developed a feature extractor tool written in Python that is able to extract different features across different categories with the aim of unifying all kinds of possible features to be extracted into one application. Such features include contextual, static technical features, and in the future, dynamic technical features.

The main focus of my part were the static technical features and I have chosen to extract features from the manifest file and sourcecode for this goal. To extract such features we mainly rely on a framework called Androguard but alternative methods exist and Fernflower is an alternative tool that we employ for sourcecode feature extraction.

Features extracted using the tool are written to CSV or JSON files. This data can be used to classify unlabeled data into benign or malicious with the use of statistics, AI, or other methods. However the purpose of this project is about feature extraction, not feature analysis so we have not focused on this part.

Finally, the tool still has several areas which can be improved on/extended on in the future. Modules of the application can be parallelized, more features extraction modules can be added, and dynamic feature extraction is another component that is not present yet.

References

[1] S. Karthick and S. Binu. Android security issues and solutions. In *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, pages 686–689, Feb 2017.

[2] T. Bläsing, L. Batyuk, A. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62, 2010.

[3] Androguard Anthony Desnos. https://github.com/androguard/androguard.

[4] dex2jar. https://github.com/pxb1988/dex2jar.

[5] JetBrains. https://github.com/jetbrains/intellij-community/tree/master/plugins/java-decompiler/engine.

[6] W. Wang, M. Zhao, Z. Gao, G. Xu, H. Xian, Y. Li, and X. Zhang. Constructing features for detecting android malicious applications: Issues, taxonomy and directions. *IEEE Access*, 7:67602–67631, 2019.

[7] Android permissions. https://developer.android.com/reference/android/manifest.permission.

[8] Android features. https://developer.android.com/guide/topics/manifest/uses-feature-element.

[9] Xiangyu-Ju. Android malware detection through permission and package. In *2014 International Conference on Wavelet Analysis and Pattern Recognition*, pages 61–65, July 2014.

[10] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69, 2012.

[11] M. Kumaran and W. Li. Lightweight malware detection based on machine learning algorithms and the android manifest file. In *2016 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pages 1–3, 2016.

[12] Alejandro Martín García, Raul Lara-Cabrera, and David Camacho. A new tool for static and dynamic android malware analysis. pages 509–516, 09 2018.

[13] 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.

[14] Vinod P. Dhanya K. A. Varsha, M. V. identification of malicious android app using manifest and opcode features. *Journal of Computer Virology and Hacking Techniques*, 05 2017.

[15] Android opcodes. https://developer.android.com/reference/dalvik/bytecode/opcodes.

[16] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild, 2018.

[17] Abdullah Talha Kabakus, İbrahim Doğru, and Aydın Çetin. Apk auditor: Permission-based android malware detection system. *Digital Investigation*, 13, 06 2015.

[18] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. 02 2014.

[19] Ravi Kiran Varma Penmatsa, Kotari Raj, and K. Raju. Android mobile security by detecting and classification of malware based on permissions using machine learning algorithms. pages 294–299, 02 2017.

[20] Ryo Sato, Daiki Chiba, and Shigeki Goto. Detecting android malware by analyzing manifest files. volume 36, page 23, 08 2013.

[21] apktool. https://ibotpeaches.github.io/apktool/.