



Cardiff University

School of Computer Science and Informatics

MSc Cyber Security and Technology Dissertation

Droidloader:
Generating New Datasets to Understand Modern
Android Malware Behaviours

Author: Alexander Hayman

Supervisor: Dr Eirini Anthi

Industry Supervisor: Dr Nia Evans

11th September 2023

Abstract

Android malware has become a prevalent threat to mobile devices with the rise in banking malware and the increase in malware capabilities. Therefore, significant research has been done on Android malware detection, especially solutions that leverage machine learning. However, most of the training data from which these models learn contain outdated Android malware, making the models less effective at detecting recent unknown malware. This thesis covers the evolution and current threats of Android malware and then explores the static and dynamic analysis methods used in research to extract key data from malware. Critical analysis is then conducted on past academic literature, including the datasets they use to see potential limitations in their research. After that, a design is created for the Droidloader tool, which can generate and analyse new Android malware datasets with careful consideration of the features to implement. The paper then discusses how the design for Droidloader was implemented, explaining certain design choices. After the tool was completed, new datasets were generated and used to plot comparisons between the features in the older datasets to see any behavioural changes. The findings show that there are changes within the malware usage of features between the new and outdated datasets. However, there were a few limitations with this approach, as there was evidence of malware type bias being present in the generated datasets.

Acknowledgement

I would like to thank Irene and Nia for supervising this project. Also, a huge thanks to PhD student Jenny, who provided tremendous support and guidance over the last few weeks of the project. I would also like to thank my family, friends, and work colleagues for encouraging me through the last few months, as I managed to push myself to the finish line with all your support!

Contents

1	Introduction	7
1.1	Project Aims and Objectives	7
2	Literature Review	8
2.1	Android Malware Overview	8
2.1.1	Deployment of Android Malware	8
2.1.2	Evolution of Android Malware	8
2.1.3	Current Threats	9
2.2	Analysis Techniques	10
2.2.1	Static Analysis	10
2.2.2	Dynamic Analysis	11
2.3	Android Malware Detection Solutions	12
2.4	Existing Android Malware Datasets	13
3	Background	15
3.1	Android Operating System	15
3.2	Application Compatibility	16
3.3	APK File Structure	16
3.4	Manifest Permissions	17
3.5	Dalvik Bytecode	17
4	Experiment Methodology	18
4.1	Choosing Android Malware Samples	18
4.1.1	VirusTotal	18
4.1.2	MalwareBazaar	19
4.1.3	VirusShare	19
4.2	Choosing Past Android Malware Dataset	19
4.3	Feature Extraction and Selection	20
4.4	Processing Results	21
4.5	Tool Design	22
4.6	Testing and Benchmarking Plan	22
4.7	Environment Setup	23
5	Experiment Implementation	25
5.1	Command Line Interface	25
5.2	Generating New Datasets	25
5.2.1	MalwareBazaar API	25
5.2.2	VirusShare and VirusTotal API	27
5.3	Extracting and Decompiling Samples	29
5.4	Extracting Features	32

5.4.1	Extracting Permissions	32
5.4.2	Extracting Classes	33
5.5	Processing Data	36
5.6	Implementation of Testing Method	39
5.7	Implementation of Benchmarking Method	41
6	Results and Evaluation	43
6.1	Generated Dataset Review	43
6.2	Testing and Benchmarking	43
6.3	Features Comparison	45
6.3.1	Permission Data Comparison	45
6.3.2	Classes Data Comparison	47
7	Conclusions	52
8	Future Work	54
9	Project Reflection	56
A	Appendix: Permissions Frequency Distribution Plots	65
A.1	MalwareBazaarRecent to CICAndMal2017	65
A.2	MalwareBazaarRecent to VirusShareRecent	66
A.3	CICAndMal2017 to CICMalDroid2020	67
B	Appendix: External, All and Android API Classes Plot Comparison	68
C	Appendix: Internal Classes Frequency Distribution Plots	70
C.1	MalwareBazaarRecent to CICAndMal2017	70
C.2	MalwareBazaarRecent to VirusShareRecent	71
C.3	CICAndMal2017 to CICMalDroid2020	72
D	Appendix: Android API Classes Frequency Distribution Plots	73
D.1	MalwareBazaarRecent to CICAndMal2017	73
D.2	MalwareBazaarRecent to VirusShareRecent	74
D.3	CICAndMal2017 to CICMalDroid2020	75
E	Appendix: Non Android API Classes Frequency Distribution Plots	76
E.1	MalwareBazaarRecent to CICAndMal2017	76
E.2	MalwareBazaarRecent to VirusShareRecent	77
E.3	CICAndMal2017 to CICMalDroid2020	78

F	Appendix: Droidloader Code	79
F.1	droidloader.py	79
F.2	core/classAnalyser.py	85
F.3	core/filterHashes.py	86
F.4	core/dataProcess.py	87
F.5	graphAnalyser.py	88
G	Appendix: Testing and Benchmarking Code	91
G.1	test_droidloader.py	91
G.2	benchmarker.py	92
G.3	core/adroguardAnalyser.py	93
H	Appendix: Droidloader Commands Options	95

List of Figures

1	Mobile banking Trojans detected by Kaspersky	9
2	Automated transfer system (ATS) process	10
3	Dynamic network packet extraction features	12
4	Android processes	15
5	Android file system	15
6	JIT and AOT differences	16
7	VirusTotal filtering hashes	18
8	AndroidManifest.xml permissions	21
9	Smali Class Method	21
10	Androguard permissions	23
11	VMware settings configuration	24
12	Droidloader command line interface	25
13	MalwareBazaar API requests	26
14	Downloading samples MalwareBazaar	26
15	Retrieving hashes by malware family	27
16	Filtering MD5 Hashes	28
17	MD5 hash sample type	28
18	Retrieving hashes by malware family	29
19	Downloading samples through VirusShare	29
20	Extracting Dataset	30
21	Decompiling dataset	30
22	APK Extraction and decompilation process	30
23	APKtool error messages	31
24	Cleaning decoded samples	32
25	Extracting permissions	32

26	Regex reluctant quantifier	33
27	Permissions data	33
28	Smali directory format	34
29	List of smali file paths	34
30	Smali class definition	34
31	Smali classname extraction	35
32	Finding invoke calls	35
33	Extracting class names from invoke calls	35
34	Extraction process output	36
35	Class name saved format	36
36	Computing frequency distribution	36
37	Processing permission data	37
38	Formatting data	37
39	Filtering Android API classes	38
40	Filtering Android API classes	38
41	Plotting Graph	39
42	Androguard Extracting Features	40
43	getPermissionsAndClasses function snippet	40
44	Validating permissions	41
45	Measuring performance	42
46	Benchmarking data	42
47	Feature extraction test results	44
48	Performance between Droidloader and Androguard	44
49	Permission frequency plot	46
50	Internal classes frequency plot	48
51	Android API classes frequency plot	49
52	Non Android API classes frequency plot	50

List of Tables

1	APK Pacakge Contents	17
2	Droidloader Commands	22
3	Experiment software requirements	24
4	Experiment hardware specification	24
5	Dataset decompilation success rate	43
6	Dataset decompilation size comparison	45

1 Introduction

Mobile devices have evolved significantly over the past decade, with an increase in computational power and more developed operating systems replacing the reliance of personal computers [1]. Applications used for mobile banking and social media have also become prominent in the mobile space. Mobile phones are often used for multifactor authentication, with two-factor short message service (SMS) authentication and authenticator applications. As a consequence, these capabilities allow mobile devices to store various sensitive data, including bank account numbers and login credentials, which can lead to financial loss and identity theft if breached. This makes it an attractive target for threat actors, especially Android devices, due to their high market share of 70.79% as of June 2023 [2]. To combat this issue, numerous machine learning (ML) based Android malware detection has been researched and developed; however, the majority of training data comes from outdated datasets [3]. As the Android malware scene continues to flourish with new capabilities and malware families appearing [4], these machine learning detection tools that are trained on older malware might not be able to detect these behavioural changes in newer malware.

1.1 Project Aims and Objectives

The main project aim is to generate a new dataset to analyse the behaviour of Android malware over the past few years. This will then be compared with previous datasets of Android malware to see the differences in their behaviour. The overall result is to determine whether the changes are significant enough, making old datasets ineffective for machine learning based Android malware detection. To evaluate the success of the project, specific and measurable goals will be set as follows:

- Understand the fundamentals of how malware is packaged, deployed, and executed within the Android ecosystem.
- Research the current threat landscape of Android malware covering recent trends, distinct malware families, and capabilities.
- Review existing literature and products on malware detection techniques and anti-virus solutions to investigate the key data used to determine if an application is malicious.
- Cover various Android reverse engineering techniques used to obtain data from a malicious Android application that can be used for further analysis.
- Generate a new dataset using various malware samples from various sources to analyse the behaviour of modern malware.
- Compare the behavioural changes of current and past malware samples to evaluate their differences.

This paper will also provide background information on the fundamentals of Android, including its operating system and the architecture of applications. Android malware are essentially

applications, so having the foundational knowledge is crucial to understand techniques malware authors often deploy to successfully exfiltrate data from victim devices.

2 Literature Review

2.1 Android Malware Overview

Having a high-level understanding of the current threat landscape of Android malware is important to investigate any new behaviours that might not be present in older malware.

2.1.1 Deployment of Android Malware

Normal malware for computers and laptops is usually deployed by tricking the user into downloading and executing the malicious file, such as phishing. However, one of the main ways for users to download applications on Android is through the Google Play Store so adversaries need to find a way to mask their malicious applications as legitimate applications. Zhou et al. [5] reports that repackaging is one of the most common techniques that malware authors use to piggyback a malicious applications into a popular application. Users are then enticed to download these applications as they do not look malicious at face value. There are also two additional techniques, the first being the update attack, which downloads the malicious payload at runtime through an update making it harder to detect. And secondly, drive-by downloads enticing users to download "interesting" apps but in reality they are malware. There have been some security measures, with Google introducing play protection in 2017 to better identify potentially harmful applications [6]. However, this does not completely stop malware from being deployed on the app store as Aleksanddr Eremin [7] discovered a number of banking Trojans that implement evil bits within the application package, so it was ignored by popular mobile security scanning tools.

2.1.2 Evolution of Android Malware

One of the first Android malware discovered was DroidSMS [8] in 2010. It worked by sending SMS messages to premium numbers so that the victim is charged money. However, these kinds of attack are harder to pull off in the present, with Android never allowing premium SMS messages by default and popular phone providers offering unlimited texts. It only took a year after the first malware developed for more sophisticated malware to appear, such as DroidKungFu, [5] which communicated with a command and control (C&C) server enabling commands to be executed remotely and supported data exfiltration. It also implemented obfuscation techniques, such as encrypting constant strings, which made it harder to reverse engineer. Malware authors were also developing techniques to hide malware with the first Android bootkit, enabling anti-virus products to evade, as it existed only in the boot partition [9]. There are numerous evasion techniques and capabilities that have been developed over the years, but these are a few notable examples from the early beginnings of Android malware.

The number of malware samples has also increased significantly each year as Symantec [10] witnessed known malware samples increase roughly four times the amount between June 2012 and June 2013. And in recent years, the number of mobile banking Trojan installers increased by 100% from 2021 to 2022 according to Kaspersky [11] although there was a dip in the year before that. The increase in malware activity shows how it is a popular avenue for threat actors to attack their target victims, leading to a greater demand for more sophisticated malware to be developed.

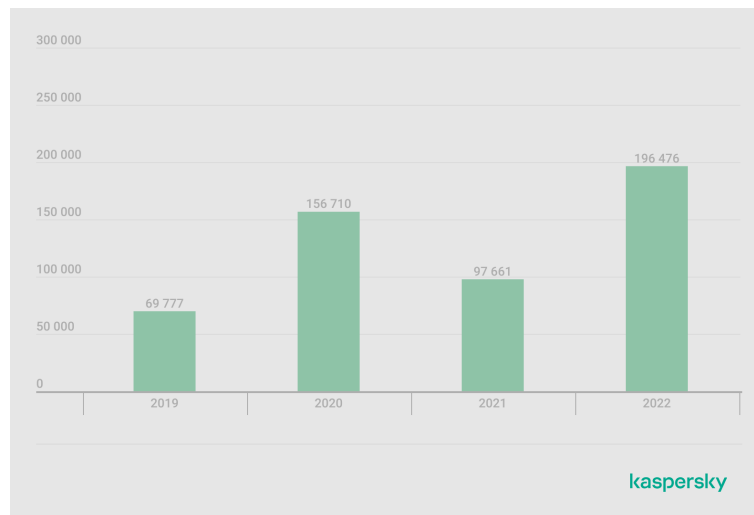


Figure 1: The number of mobile banking Trojans detected by Kaspersky in 2019-2022 [11]

Malware is often sold on the dark web where an attacker pays someone else who is an expert at deploying malware onto the Google play store. Kaspersky [12] found out that a price of a loader service, the process of injecting malicious code into a Google play store, is commonly priced between \$2000 and \$20,000 . Overall, the rapid development of Android malware and its increase in activity over the past decade shows how older malware can quickly become irrelevant; malware authors will try to implement new and innovative features as a unique selling point to sell their malware. Therefore, creating new Android malware datasets is crucial to provide effective training data for malware detection as its more likely to contain the characteristics of modern malware.

2.1.3 Current Threats

Banking malware has been a significant threat in the recent year with the appearance of new malware families and twice as many malicious droppers on official application stores according to ThreatFabric [13]. Mobile banking has become prevalent in society, where 89% of the survey respondents said they use mobile banking according to the Intelligence Mobile Banking Competitive Edge Study [14]. The surge in the use of banking apps attracts threat actors to find novel ways to earn money by stealing directly from victims rather than selling their data online. ThreatFabric also discovered a new capability called an automated transfer system (ATS), which first appeared in 2019 as a prototype and then in 2021 where it was fully capable of achieving its objective.

This technique enables criminals to automate the whole attack chain, from an infected device to fully compromising the victim's bank account. It leverages accessibility logging, which is the extensive logging of all UI elements and all the events associated with them, such as clicking and swiping.

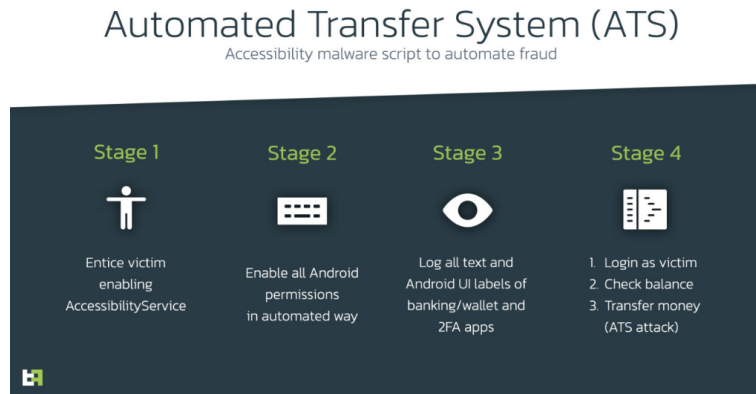


Figure 2: Automated transfer system (ATS) process [13]

There are also other techniques deployed other than ATS such as 2FA code stealer and the exfiltration of seed phrases from cryptocurrency wallets. These new techniques would not have been covered in the datasets prior to the first appearance of ATS. Therefore, it would be more difficult to detect this attack if the malware detection framework were trained using an older malware dataset, as it would not have looked for these specific attack vectors.

2.2 Analysis Techniques

Reviewing analysis techniques used in the past literature provides information on the data that is normally extracted from Android malware samples, as well as the advantages and disadvantages of certain analysis techniques.

2.2.1 Static Analysis

Static analysis is a technique that examines a malware sample without executing it. Different static features such as permissions, developer ID, API calls, intents, and components have been used for malware detection [15]. Permissions are one of the most investigated characteristics, used for various Android malware detection frameworks with Gyunka et al. [16] extracting permissions to produce data that can be used to fit different learning classifiers. They are quick to process, with tools such as Droidloader [17] checking for suspicious permissions to filter out a large collection of Android applications.

Another feature that is continuously extracted for analysis is API calls, as they provide information on a set of methods that a malicious application might use as part of their attack chain. API calls are essentially class methods that are part of the Android package [18], with Google providing documentation for the package's classes and methods. These can be leveraged

to look for suspicious calls that perform malicious tasks. In the intelligent mobile malware detection paper [19], API calls were extracted from a sample of 27,891 containing malicious and benign applications. The set of API calls for each application was observed to indicate malicious behaviour. For example, the API calls `Android/telephony/SmsManager;.sendTextMessage` and `Android/telephony/SmsManager;.getDefault` would indicate an SMS attack. Based on API calls, applications are then categorised into distinct groups (Ambiguous, Risk, and Disruptive). More advanced frameworks, such as Kunai [20], offer a Dalvik parser for accessing the DEX structure allowing more features to be extracted, namely cross references(XREF). This can be used to filter out method calls by checking if they are referenced in the main application rather than a third-party package.

Publicly available tools such as Jadx [21] assist in static analysis by converting the dex code to Java to easily analyse the decompiled code; however, Jadx cannot be automated and is therefore not suitable for processing a large amount of samples. In all of these examples, a large number of samples were processed, so the main advantage of this technique is that it can quickly analyse the behaviours of thousands of malware samples.

2.2.2 Dynamic Analysis

Dynamic analysis is the process of running a malware in a safe environment called a sandbox to analyse its true behaviour. Researchers have found techniques for automated dynamic analysis for detection [22], so it could be a suitable choice for processing large Android samples. Tools such as Monkey Runner [23] can aid researchers by simulating events on the mobile screen to capture more data. One of the most frequently extracted dynamic features is system calls, which are calls offered by the kernel to access hardware resources. Zhang et al. [24] analysed system call traces to produce unique system call sequences and observed a difference where a specific sequence appeared more in malicious traces than in benign traces, and vice versa. The collection of system call traces was automated using Genymotion [25], an Android emulator, and the Android debug bridge (adb) [26] command line tool that allows desktop computers to communicate with mobile devices. Another notable feature observed is the traffic on the network, in particular the TCP, HTTP, and DNS packet features. The network-based indicators shown in the network packet table by Shyong et al. [27] (Figure 3) show some of the extracted features. CREDROID [28] proposed a solution to inspect the contents of packets for personally identifiable information (PII) by searching for keywords such as passwords and location. The solution also observed URL data using the Web Of Trust API to determine the reputational score of the URL with a poor rating, suggesting that the website contains some identity theft risks.

However, some malware will try to employ preventive techniques to avoid being dynamically analysed. An Android malware hardening survey [29] shows numerous techniques being used, such as anti-emulation and anti-debugger; mobile malware can judge whether its running on a real device or not, based on the differences such as system properties, sensors, data, and network between the real device and emulator [30]. In general, dynamic analysis can extract features that are not present in static analysis however, it requires running malicious samples on a device and

attempting to avoid anti-analysis techniques which could be an extensive process depending on the size of the dataset to process.

Network Traffic Types	Features
TCP	<ul style="list-style-type: none"> ● Uploading / downloading packet number ● Uploading / downloading bytes ● Average bytes of uploading / downloading ● Connected IP
HTTP	<ul style="list-style-type: none"> ● HOST ● Request-Uri ● Request-Method ● User-Agent
DNS	<ul style="list-style-type: none"> ● Domain name
Other	<ul style="list-style-type: none"> ● Leakage of privacy information ● IP Reputation Score

Figure 3: Dynamic network packet extraction features [27]

2.3 Android Malware Detection Solutions

The detection of malware in Android applications based on machine learning has been increasingly used by researchers, as it can learn more advanced malware traits compared to signature-based anti-malware [3]. This approach requires a dataset in order to learn and determine whether an unknown application is benign or malware. Datasets used in machine learning are different, as they contain a mixture of malware and benign application instead of just malware. However, finding malware proves to be a tricky task for researchers, as the applications on the market are more likely to be benign; several papers end up using many more benign files than malware as a result of this issue [31], [32]. The issue can be resolved by using existing malware datasets such as Drebin [33] to add malware files to their dataset. To see the effect of not using recently generated malware datasets, it is important to review previous research to identify any limitations in their approach.

Han et al. [34] used the Drebin dataset as part of their training data for their machine learning approach to detect Android malware based on API calls. It reported an impressive accuracy rate of 99.75% in identifying if an application was malicious or benign. But this test was carried out with malware in the Drebin dataset, and therefore the accuracy score does not indicate its effectiveness against current Android malware. However, Fan et al. [35] uses a different approach to download malware from various sources, such as malware from 2013 to 2014 on VirusShare [36] and malware from MassVet [37], which are then uploaded to Virustotal [38] to classify the malware into unique malware families. This can then be used to detect unknown malware, which was tested on recent and old malware samples. It could detect 97.8% old malware from a sample of malicious and benign applications, but only 75% when detecting recent malware. Although various datasets were used, the accuracy rate still decreased due to the fundamental concept drift problem [39], where models built through training on older malware can make poor and

ambiguous decisions when faced with modern malware. There are many more cases of studies using outdated datasets according to muzaffar et al. [3] and, as a consequence, their effectiveness deters with newer malware.

2.4 Existing Android Malware Datasets

Reviewing key Android malware datasets used in previous studies can give insight into the various collection methods used by researchers to retrieve malware. Knowing how malware is collected can identify any flaws in their processes that this experiment can solve.

Android Genome Project and Drebin Datasets

The first dataset is from the Android Genome Project [5] containing 1260 Android malware samples ranging from August 2010 to October 2011. They obtain the list of malware by examining related security announcements, threat reports, and blog content from existing mobile anti-virus companies and active researchers at the time. Unfortunately, all the research content used by the project to find malware is not accessible, so the sources cannot be evaluated. The list of malware was then downloaded by requesting malware samples from researchers or actively crawling from the existing official and alternative Android application market. Overall, this approach would be impractical given that the number of malware installers discovered each year currently has reached more than a million samples; it would be impossible for researchers and companies to produce work that exhaustively examines most malware today. However, it excels at removing false positives in the dataset by running the malware through numerous mobile anti-virus software, such as AVG Antivirus Free v2.9 or Norton Mobile Security Lite v2.5.0.379. If the anti-virus software did not pick it up, the sample was manually verified to confirm its malware.

The Drebin dataset was produced and shared as a result of an Android malware detection study [33] using broad static analysis. It contains 5560 malware applications collected from August 2010 to October 2012 by downloading and processing applications from the GooglePlay store, Chinese markets, Russian markets, and other sources such as Android websites and malware forums. To determine if an application was malicious, it was sent to VirusTotal [38] where the output was inspected from the anti-virus scanners. All applications were classified as malicious if they were detected by at least two of the scanners, and all samples classified as adware were removed. The dataset also includes all samples from the Android Genome Project [5]. This method is more scalable compared to the Genome project as it does not actively search for malware for the majority of the data but instead downloads a large collection of samples from application markets and filters for malware through VirusTotal which could be automated. Using various anti-virus scanners instead of one also reduces the chances of false positives.

CICAndMal2017 and CICMalDroid2020 Datasets

The CICAndMal2017 dataset is a more recent dataset generated to cover the shortcomings and limitations of previous datasets [40]. Although not well established compared to previous datasets, it is publicly available to download and contains 426 samples with 42 unique malware families. It collects samples from several sources, such as VirusTotal [38] and Contagio Security blog [41] but the time period for when the malware was collected is not specified. Some of the samples were also removed due to sample error (unsigned and corrupted apps) and inconsistent malware labelling, where the malware family is inconsistent among both academic and industry fields. False positives were also removed by only accepting malware that is flagged by more than two antivirus products in Virustotal. The advantage of using this dataset is that it labels malware into four categories: Adware, Scareware, Ransomware, and SMS malware allowing flexibility in choosing the type of malware to analyse. However, the paper does not explain how the malware was categorised.

Finally, the CICMalDroid2020 dataset [42] is the largest with 9803 malware samples generated to solve the problem of classifying categories for unknown malware. The sample collection is roughly the same as the process for the CICAndMal2017 dataset, but also includes the AMD dataset and other datasets used by recent research contributions [43], [44]. Samples are collected from December 2017 to December 2018 but this collection process did not check for false positives. The categorisation of malware is slightly different, with malware categorised as follows: Adware, Banking Malware, SMS Malware, and Mobile Malware. 17,341 samples were initially collected, but were filtered if they did not run successfully on CooperDroid [22] due to different error types, such as bad ASCII characters or bad unpack parameters. After processing the samples, only 11,598 applications remained, with 9,803 samples identified as malicious and the rest benign.

In general, reviewing the specified datasets shows how all the datasets rely on previous research or datasets to collect malware samples, although the datasets used were the most recent during the collection time period. The main issue is that none of the datasets specify a tool or an automated procedure for collecting malware, so the process cannot be replicated for newer malware. The program that will be created in the project aims to solve this problem. Despite this, most datasets have a standard test for false positives to further validate that their dataset contains correctly labelled malware. All datasets except the Genome Project dataset leveraged Virustotal to either download malware or as an anti-virus scanner, and so this particular service should be used in the experiment. Malware categorisation should also be considered, as the CICAndMal2017 and CICMalDroid2020 datasets categorised their malware, but their respective articles did not define how they were categorised.

3 Background

Android operating system differs greatly in terms of how applications run and interact with the system compared to standard desktop operating systems such as Windows or Ubuntu. As it provides a high level of abstraction, most of the technical details of how applications are compiled, installed, and run on devices are hidden through the normal user experience. This section reveals the hidden technicalities behind the Android OS and its supported applications.

3.1 Android Operating System

The Android OS has two main components: a modified Linux kernel and a modified Java virtual machine (JVM) optimised to run mobile applications. By comparing the processes and file system between a conventional Linux operating system, the differences are apparent.

```
root@generic_x86:/ # ps
USER      PID   PPID  VSIZE  RSS   WCHAN          PC  NAME
...
u0_a9      3785   2025 1335328 72612 SyS_epoll_ b72ce685 S com.android.launcher3
u0_a52     3824   2025 1277828 40992 SyS_epoll_ b72ce685 S com.android.messaging
u0_a47     4050   2025 1297964 40112 SyS_epoll_ b72ce685 S com.google.android.gm
u0_a46     4285   2025 1302476 38988 SyS_epoll_ b72ce685 S com.google.android.play.games.ui
u0_a33     4391   2025 1262900 25720 SyS_epoll_ b72ce685 S com.google.process.gapps
u0_a4      4830   2025 1259740 25628 SyS_epoll_ b72ce685 S com.android.defcontainer
...
```

Figure 4: Android processes

The list of processes on the Nexus 5 Android 6.0 emulated device shows how each application is running as a different user compared to a standard Linux desktop where applications are usually run under the user who started the application.

```
root@generic_x86:/ # ls
acct
cache
charger
config
d
data
default.prop
...
```

Figure 5: Android file system

The root directory also contains additional folders/files specific to the Android device that are not common within a normal Linux desktop file system. These are only a few notable examples with numerous other comparisons that could be made to show their differences.

The modified JVM for Android has gone through two main iterations, Android Runtime (ART) and its predecessor Java Dalvik virtual machine (DVM) used in Android 4.4 and below. ART offers performance increases using ahead-of-time (AOT) compilation where all the code is compiled during the installation of the app instead of during runtime as shown in Figure 6. This has its drawbacks as compiling code increases both the installation time and storage required for the application. However, both runtimes are compatible with executing Dalvik Executable (Dex) format and Dex bytecode specification [45], which is the compiled format for Android applications.

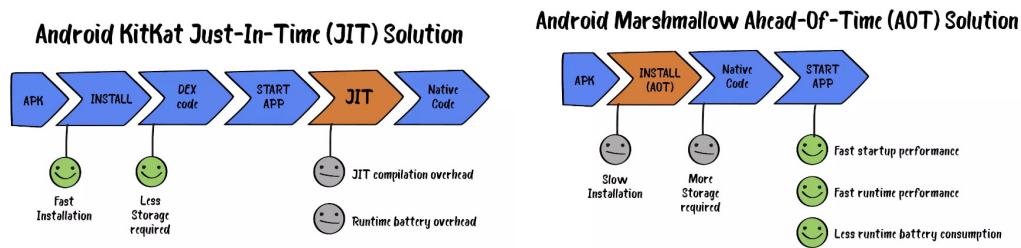


Figure 6: JIT and AOT differences [46]

3.2 Application Compatibility

With a rapidly developing operating system, it is important to understand the behaviour between applications and major Android releases as it can affect its compatibility. Each major release updates the existing Android application programming interface (API), with the addition, modification, and removal of packages [47]. The Android API is used to interact with the underlying Android system, with the initial release being API level 1 for Android 1.0, and subsequent releases have incremented the API level. Google [48] has made sure that applications are forward compatible with newer versions of Android where most changes are additive, except for isolated cases where an application uses a part of the API that is later removed for a specific reason. However, the same cannot be said for backward compatibility because previous versions of the Android platform do not include the new APIs. This could affect the behaviour of recent malware as they use packages that are not present in older malware samples.

3.3 APK File Structure

The Android Package (APK) is a ZIP archive file with the .apk file extension and is used to store and distribute Android applications. These packages contain all the relevant compiled code, resources, and metadata required for a complete application. Since this is the only file that is given when analysing Android malware samples, it is important to understand the defined files and folders from an extracted APK. The following table below describes each defined entry:

Entry	Description
AndroidManifest.xml	Contains the configuration information on the application and the defined security parameters, such as required permissions, name, and version.
classes.dex	The application code compiled in the Dex format.
resources.asrc	A table file containing a record of the mappings between the resource file and the related resource ID.
assets/	A folder containing application assets that can be retrieved by AssetManager.
lib/	Contains native libraries that are included with the application.
META-INF/	The signature of the APK is stored in this folder, as well as the list of all files contained in the zip archive and their hash values.
res/	This contains any files primarily media files used for the application, such as images or activity layouts.

Table 1: APK Package Contents

3.4 Manifest Permissions

By default, Android prevents applications that require access to restricted data or actions unless the appropriate permissions are declared in the AndroidManifest.xml file [49]. Therefore, sophisticated malware must trick users into accepting dangerous permission so that it can gain further control and access to the victim's phone and data. Knowing how these permissions work allows for further investigation into dangerous permissions. First, Android has different types of permissions such as install-time, normal, and signature permissions [50]. These permissions give applications limited access to restricted data and present little risk to user's privacy. On the other hand, there are dangerous permission types such as runtime permissions, granting applications access to restricted data or actions that attackers could take advantage of. This is one of the key permission types to further analyse, since malware applications will normally abuse runtime permissions to gain as much access to the phone as possible.

3.5 Dalvik Bytecode

All the application's logic is stored in the classes.dex file, so knowing a few of the basic bytecode instructions can give a general idea of what classes and methods the applications are using. Reading the classes.dex file to get meaningful results is often difficult, as it is just a sequence of hex characters. Using tools such as Apktool allows the decompilation of the application logic into Smali code resulting in human-readable code. Google has provided documentation specifying the description and syntax for each bytecode [51].

4 Experiment Methodology

4.1 Choosing Android Malware Samples

There are several websites that contain recent malware samples that can be used to create new datasets such as MalwareBazaar [52]. One of the key requirements is a way to fetch the download through an API so that the generation of new datasets can be done automatically. It should also be dynamic, meaning the tool will try to obtain the most recent samples from websites hosting the malware to generate a dataset that is reflective of current malware. There were other methods that were considered, for example, using an existing and regularly updated dataset such as Androzoo. However, this dataset was too large, contains benign samples and dates back to 2016 requiring a vast amount of pre-processing to extract the relevant samples.

4.1.1 VirusTotal

VirusTotal [38] is one of the most popular virus analyser websites where users can upload and query samples to check if they are malicious. It also has an API however, users without an enterprise licence are limited to the amount of endpoints they can query. The Advanced corpus search endpoint [53] allows searching for files so in this case, the endpoint can be leveraged to find files with the APK extension, but this feature is only available to enterprise users. The only use case that can be used with VirusTotal is filtering a list of general malware hashes to only filter for malware hashes that relate to Android as shown in Figure 7.



Figure 7: Filtering hashes through VirusTotal

The <https://www.virustotal.com/api/v3/files/hash> endpoint retrieves the file report that matches the hash value, and then the program can use it to check if the file is an Android APK file. The daily quota for API requests is limited to 20,000 requests for academic use, so it may take more than a few days to get a sufficient dataset size that contains Android malware.

4.1.2 MalwareBazaar

MalwareBazaar [52] is a project from abuse.sh with the aim of sharing malware samples with the cyber security community. MalwareBazaar does not accept adware and has a strict malware submission policy [54], meaning that the chances of false positive malware are very slim. It also offers an API with a wider range of endpoints offered compared to VirusTotal. It can query samples by filetype, but the maximum limit of hashes to retrieve is 1,000 hashes. As a consequence, it can only retrieve and download 1,000 of the most recent samples, since the query cannot specify a range or sort order. MalwareBazaar also labels malware with specific signatures, such as Cerberos, but does not document the process of how it classifies malware to a specific signature, so it should be viewed with some caution.

4.1.3 VirusShare

The last virus hosting platform to consider is VirusShare [36]. Compared to MalwareBazaar, it contains samples of general malware and implements an API for programmes to interact with the website. Sending a hash of a sample through the API returns a report similar to VirusTotal, containing key information such as file type to allow for further filtering. However, the API has limited functionality, and requests are limited to 4 requests per minute [55]. Despite that, the website also hosts a repository of APK samples and a list of MD5 hashes for all malware samples on the website. Unfortunately, the latest collection of APK samples is too large for sufficient processing, as the size is roughly 200GB, storing 20,000 samples. The only option is to resort to the list of MD5 hashes and leverage the VirusTotal API to filter for Android specific malware. One of the issues with VirusShare is that it does not document how malware is uploaded to the site, so some precaution should be taken for false positive malware. However, since samples are being sent to VirusTotal for filtering, they can be verified as malicious by observing how many anti-virus scanners detect the sample.

4.2 Choosing Past Android Malware Dataset

An appropriate Android malware dataset must be chosen to compare it with the generated dataset from the experiment. The key requirements are that they are not too large (over 100GB) and date back at least five years. One of the issues is that some datasets used in academic papers for training data and analysis have either been discontinued or require permissions to access the dataset. For example, the website hosting the AMD dataset was unavailable during the time period of this project. Both the Drebin and Android malware genome datasets were also inaccessible during the project period. As a consequence, the experiment will have to use the CICAndMal2017 and CICMalDroid2020 datasets, which have not been extensively used in past academic work. Despite this, both datasets provide sufficient documentation of how they collected their malware, which makes them valid datasets to use in the experiment. Any benign application contained in the datasets will be ignored as the project goal is malware analysis. Adware will also be removed

so that valid comparisons can be made with malware from MalwareBazaar, which does not contain Adware.

4.3 Feature Extraction and Selection

To measure and compare the new dataset created during the experiment with the previous dataset, features must be extracted from both datasets. A methodology must also be created to automate the extraction of key information from a collection of APK files. There are many features that could be extracted from an APK file from both static and dynamic analysis, but only a few were selected to keep the experiment in scope. The experiment will only use static analysis methods, as it is faster at extracting data from a large number of malware samples compared to dynamic analysis; the application does not need to run on a device, which takes longer to analyse, and it could potentially be harder to automate. The selected features to extract were permissions and classes, since various malware samples share the same set of classes and permissions, making it measurable to compare their frequency between two datasets. Comparisons between classes contained in the Android API package [18] can also be compared between two datasets.

Decompiling APKs

As APK files are essentially a ZIP packaged file, it can be extracted by using an unzipping tool; however, there are a few limitations to this approach. First, some of the files are unreadable, including AndroidManifest.xml, and there is no decompilation, which makes it challenging to understand the application logic. A more suitable tool would be Apktool [56], which allows readable files to be extracted. Apktool was designed with the main purpose of disassembling, assembling and analysing APK files. It also decompiles Dalvik bytecode into smali code, which will be useful when extracting classes. It has been around for more than thirteen years maintaining a strong open source community and requires minimal installation and prerequisite knowledge. It has also been used in academic work with Gunkya et al. [16] using it to reverse engineer Android APKs.

Extracting Permission Data

The permission information for each Android application is defined in the AndroidManifest.xml file, which can be read after decoding a sample with Apktool. Figure 8 shows the format of defining permissions with the use of <uses-permission> tag. A description of each permission is provided in Google's Android developers documentation [49]. To extract only permissions, regular expressions (regex) can be used to find all instances of the <uses-permissions> and then retrieve the permission name from inside the quotations.

```

1 <uses-permission android:name="android.permission.INTERNET"/>
2 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
3 <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
4 <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
5 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
6

```

Figure 8: AndroidManifest.xml permissions example

Extracting Classes Data

Android applications are built with languages that follow the object-orientated paradigm (Kotlin or Java). Google provides packages containing Android specific classes and methods to provide a layer abstraction from the internal mechanisms of the operating system. This is so that developers can easily interact with the operating system when building applications, saving a significant amount of time. Developers can also import standard Java classes and third-party libraries externally or internally. Extracting classes from a dataset of malicious APKs is useful to investigate as it can give insights into any common methods the applications are leveraging to execute malicious commands. One of the methods to extract the classes is by inspecting the smali code to look for specific calls. These are five main calls to look for when finding method calls which include: invoke-virtual, invoke-super, invoke-direct, invoke-static and invoke-interface. Figure 9 shows an example of the Android class PrintedPdfDocument being closed. Regex can also be used to extract the class names from the invoke commands.

```

1 invoke-virtual {p1}, Landroid/print/pdf/PrintedPdfDocument; -> close ()V
2

```

Figure 9: Invoking class method in smali

4.4 Processing Results

After extracting the mentioned features from a dataset, it will be processed to produce relevant results to compare the behaviour between two Android malware datasets. The main comparison to make is the frequency of permissions and classes used in both datasets to see which ones were the most common to use in Android malware during the year the dataset was generated. The experiment will present these data in a horizontal double-bar chart of two Android malware datasets and the categories would be the top n most common features used in a dataset, where n is the number of unique features. This is to visually see the differences in the frequency of features extracted, which could indicate behaviour changes in both datasets.

4.5 Tool Design

As performance of the tool is not main factor for this experiment, the tool will be programmed in Python due to performance not being a main factor in the experiment, as well as support for numerous files/folder manipulation with Python modules such as the os module [57]. The name given to this tool is Droidloader, which will perform various functions and will have a built-in command line interface to provide flexibility in what commands to execute for a target dataset. To avoid scope creep of the project, a set of required commands will be specified to measure the completion of the program.

Command	Description
Download	Download malware samples to a folder to generate a new dataset by specifying which website to download the files.
Extract	Samples downloaded from the website are usually encrypted in a zip folder. This command extracts the APK from the zip folder to be left with a collection of malware samples contained in a folder.
Decompile	This command decompiles a dataset containing APKs using the Apktool. The decompiled dataset is saved to a separate folder for further analysis.
Filter	Filter the list of MD5 hashes of general malware samples provided by VirusShare to generate a list of MD5 hashes of Android specific malware. This list can then be used to download samples which have the same MD5 hash value.
Permissions	Extract the permissions from a decompiled Android malware dataset.
Classes	Extract classes from the decompiled dataset of Android malware dataset.
Graph	Produce a bar chart of permissions or classes to visually view the differences between two datasets.

Table 2: Droidloader Commands

4.6 Testing and Benchmarking Plan

Droidloader performs two main functions, generating the dataset and then extracting features from a dataset. The first function is simple to test, as the tool should output a folder with a batch of APK files. The second function is more difficult as there needs to be a verification check to determine if the features extracted from the dataset are correct. As the experiment is mainly focused on the results, rather than on the tool itself, a set of thorough test cases for the tool will not be produced to check for edge cases.

Testing with Androguard

Androguard is a reverse engineering tool to analyse Android files [58] and can extract various key information about an APK, for example, permission data as shown in Figure 10. It has the reputation of being a sufficient application, as it has been explored in past academic papers; for

example, DroidBot [59], a test input generator for Android, and popular embedded malware analysis frameworks such as Mobile Security Framework (MobSF) [60].

```
1 In [2]: a.get_permissions()
2 Out [2]:
3 ['android.permission.INTERNET',
4  'android.permission.WRITE_EXTERNAL_STORAGE',
5  'android.permission.ACCESS_WIFI_STATE',
6  'android.permission.ACCESS_NETWORK_STATE']
7
```

Figure 10: Androguard extracting permissions example

Androguard can also take the form of a Python package, where it can be embedded in existing applications. This tool will be used to compare the features extracted from the experiment to verify that the features are correct. The plan for testing would be to generate a random sample of 10 malicious applications and verify whether the permissions and classes extracted from Droidloader are the same as those extracted from Androguard.

Measuring Performance

Performance was not the main goal for the experiment; however, to justify building a tool from scratch instead of utilising third-party tools, performance must be measured between Androguard and Droidloader to determine if Droidloader performs better. The main performance factor to consider is speed, as it is more favourable to use a tool that can process a dataset quicker than the other. This will be done by creating a benchmarking plan consisting of three randomly generated datasets containing 10, 25 and 50 random samples and then measuring the time it takes for both tools to extract the classes and permissions of each dataset. The reason for using various dataset sizes will be to determine whether the tools would perform better or worse if they had more data to process. As performance can slightly vary due to external factors that cannot be controlled (operating system processes), the performance test will be done five times, where an average will be taken.

4.7 Environment Setup

As malware will be downloaded to a machine during the experiment, using a virtual machine would be a sufficient barrier to negate any damages if the malware accidentally executes on the machine. Although APK files are incompatible with a desktop operating system, there could be accidental download of malware compatible with the host operating system. Figure 11 shows the virtual machine settings that will be used to conduct the experiment. Using a separate host machine only containing the malware samples would not be necessary as the risk of malware being accidentally executed is negligible. Table 3 and Table 4 shows the software requirements and hardware specifics that are going to be used to run and test the experiment.

Software Version	Description
Vmware Workstation Pro 17	A desktop hypervisor software to run virtual machines.
Ubuntu 22.04.3 LTS Desktop	A linux distribution to host the experiment. As there are no specific dependencies to install, this distribution would be sufficient enough to run the experiment.
Python 3.10.12	To run DroidLoader which is created in the experiment.
Apktool 2.8.1	To decompile the APK files to extract readable files
Androguard 3.3.5	Testing and benchmarking the tool created in the experiment
OpenJDK 11	To run Apktool as it is a Java Archive (JAR) file

Table 3: Experiment software requirements

Item	Specification
Processor (CPU)	AMD Ryzen 7 5800H
Memory (RAM)	16GB
Disk Storage (SSD)	475GB
Graphics Processor (GPU)	NVIDIA GeForce RTX 3060

Table 4: Experiment hardware specification

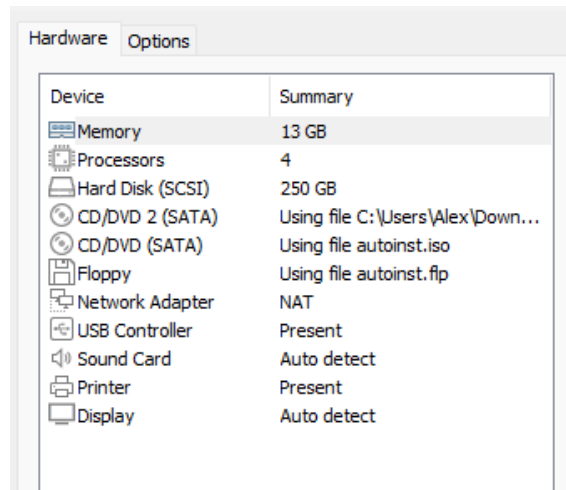


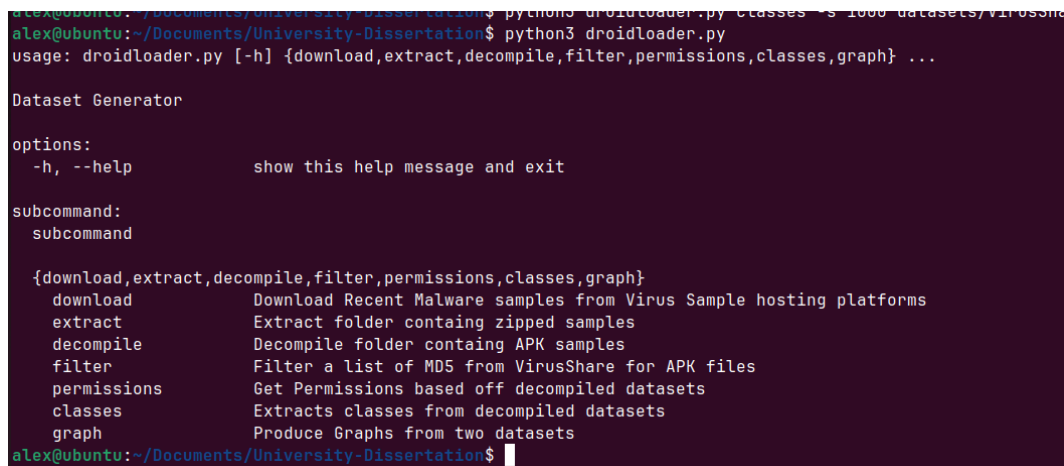
Figure 11: VMware settings configuration

5 Experiment Implementation

This section focuses primarily on the logical aspect and design choices for each main component of Droidloader, instead of explaining Python's syntax in detail. Appendix F shows the full code base for the Droidloader tool.

5.1 Command Line Interface

Using the argparse module [61], the program can perform a particular action, depending on the arguments set by the user. The first argument is the main commands the user can use, as shown in Figure 12. Each main command then has additional arguments and command line options, which are different depending on which command is used. Appendix H shows screenshots of each of the main commands, including their options.



```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py classes -s 1000 datasets/VirusShare
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py
usage: droidloader.py [-h] {download,extract,decompile,filter,permissions,classes,graph} ...

Dataset Generator

options:
  -h, --help            show this help message and exit

subcommand:
  subcommand

{download,extract,decompile,filter,permissions,classes,graph}
  download              Download Recent Malware samples from Virus Sample hosting platforms
  extract               Extract folder containing zipped samples
  decompile             Decompile folder containing APK samples
  filter                Filter a list of MD5 from VirusShare for APK files
  permissions           Get Permissions based off decompiled datasets
  classes               Extracts classes from decompiled datasets
  graph                Produce Graphs from two datasets
alex@ubuntu:~/Documents/University-Dissertation$
```

Figure 12: Droidloader command line interface

5.2 Generating New Datasets

5.2.1 MalwareBazaar API

The MalwareBazaar API only has one endpoint which is <https://mb-api.abuse.ch/api/v1/>; however, modifying the post-form data can return different results depending on the value of query. The first query value is `get_file_type` allowing samples to be searched with a specific type but it only returns information about the sample and not the binary data of the sample. However, one of these data points that is returned is the `sha256_hash` which can then be used to download the sample by specifying the query value to be `get_file` and providing the `sha256_hash`.

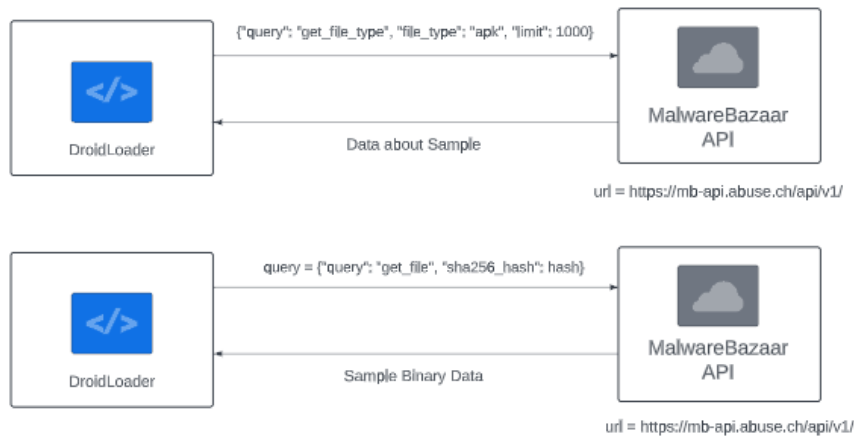


Figure 13: MalwareBazaar API requests

The limit parameter value can also be specified to specify how many samples to return from the query with the maximum value being a 1000 samples. Droidloader allows the user to specify the number of samples to download as long as its below the maximum limit. The binary data is then written to a folder as shown in Figure 14.

```

1 def downloadSample(dir, hash):
2     query = {"query": "get_file", "sha256_hash": hash}
3     response = requests.post(MBURL, data=query, timeout=60,
4                             allow_redirects=True)
5     open(f"{dir}/{hash}.zip", "wb").write(response.content)

```

Figure 14: Downloading samples through MalwareBazaar

The API also allows us to search for files by the type of malware family to which they belong using the get_siginfo query value. The getMalwareByFamily function returns a list of hashes by providing a list of notable malware families such as Hyrda or Cerberus. This is only an experimental feature and will not be used when generating a dataset for analysis. As mentioned above, MalwareBazaar does not document how they classify malware into a particular family, so it will not be valid to use as a comparison dataset.

```

1 def getMalwareByFamily(families, limit):
2     hashes = []
3     for family in families:
4         query = {"query": "get_siginfo", "signature": family, "limit":
5             limit}
6         hashes.extend(getHash(query))
7     return hashes

```

Figure 15: Retrieving hashes by malware family

5.2.2 VirusShare and VirusTotal API

Downloading samples using both of these platforms requires a workaround due to a request limit and limited endpoints. VirusShare provides a list of files in the format of a basic text file, each containing 65,536 MD5 hashes. The text file can then be used to send GET requests to <https://www.virustotal.com/api/v3/files/{id}> substituting the id parameter to be the MD5 value; the endpoint then returns a file object when the request is valid. This file object contains the `type_description` and `type_tag` attributes, which can then determine whether the MD5 value belongs to an Android malware sample (Figure 17). It also contains the `malicious` attribute which totals the amount of anti-virus scanners that marked the sample as malicious. If it is picked up by more than two scanners, it is deemed malicious. As the daily request limit for VirusTotal is 20,000 a day, requests must be spread over four days to filter APK samples from one MD5 file.

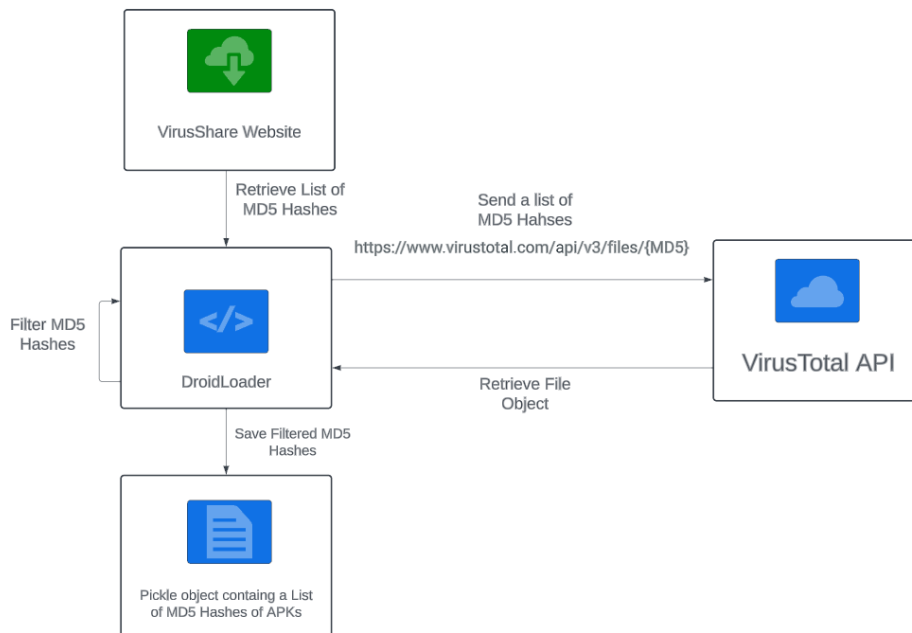


Figure 16: Filtering MD5 hashes

```

1 sampleType = response.json()['data']['attributes']['type_description']
2 extension = response.json()['data']['attributes']['type_extension']
3 detections = response.json()['data']['attributes']['last_analysis_stats']
4               ['malicious']
5 if sampleType == "Android" and extension == "apk" and detections >= 2:
6     return hash
7 else:
8     return None

```

Figure 17: Checking MD5 hash sample type

The pickle module is used to keep track of MD5 hashes that Droidloader has processed, allowing the serialisation and deserialisation of the python object so that it can be saved to disk [62]. Figure 18 shows the format of the pickle object where it contains the list of MD5 hashes and the count value showing the number of MD5 hashes that have been processed by VirusTotal.

```

1 import pickle
2 with open("VirusShare_00476.pkl", "rb") as f:
3     print(pickle.load(f))
4 ...
5 {'hashes': ['675bda8cff31499260a41805de3bc667', ...
6 '49c2416e0c88fd129b481c2aa646d9da'], 'count': 38000}
7

```

Figure 18: Retrieving hashes by malware family

After processing the MD5 file, the list of hashes can be sent to <https://virusshare.com/apiv2/download> endpoint with a GET request specifying the hash value as one of the parameters. Requests are limited to 4 per minute, so only 240 samples can be downloaded per hour. Overall, this method is significantly longer than using MalwareBazaar as it takes days to process; however, the trade-off of being able to generate a larger dataset justifies the implementation of this method.

```

1 def downloadSampleVS(dir, hash, limit):
2     count = 0
3     if count < limit:
4         time.sleep(15)
5         url = f"https://virusshare.com/apiv2/download?apikey={VSAPIKEY}&
6 hash={hash}"
7         response = requests.get(url)
8         open(f"{dir}/{hash}.zip", "wb").write(response.content)
9         count+=1

```

Figure 19: Downloading samples through VirusShare

5.3 Extracting and Decompiling Samples

The samples downloaded from both methods generate a new dataset, but they are not in the correct format as they are stored in a ZIP file. By processing the datasets, it can be stored in the correct format and allows features to be extracted from the dataset. When a sample is downloaded from either MalwareBazaar or VirusShare, it contains an encrypted ZIP file with the passphrase "infected", so it needs to be extracted. This can be automated with Python by extracting each ZIP file from a specified folder to a different folder, resulting in a valid dataset that contains only APK files as shown in Figure 20. However, extracting from VirusShare samples does not put the APK extension for the extracted file, so Droidloader needs to add the extension in order to format the sample correctly.

```

1 def extract(targetdir, outputdir):
2     os.mkdir(outputdir)
3     for archive in os.listdir(targetdir):
4         if archive.endswith(".zip"):
5             with pyzipper.AESZipFile(f"{targetdir}/{archive}") as zf:
6                 zf.extractall(path=outputdir, pwd=bytes("infected", "utf
7                 -8"))

```

Figure 20: Extracting Dataset containing ZIP files

Once the datasets are formatted correctly, the APKtool command line program can be used to decode each file ready for further analysis. This process can also be automated by taking advantage of the `os.system` function, which allows the execution of Linux commands within Python. One dataset such as the CICAndMal2017 dataset contains sub directories so the program needs to crawl through each directory and then store the absolute path of each APK file into a list. The list is then iterated by running the APKtool command as shown in Figure 21.

```

1 for sample in samples:
2     if sample.endswith(".apk"):
3         os.system(f"cd {outputdir} && java -jar {apktool} d {sample}")
4

```

Figure 21: Decompiling dataset containing APK files

The output of the decompilation is written in a new directory where the user can specify the name of the new directory. This is to avoid tampering with the original dataset in case it needs to be reused. Java and Apktool must be installed before decompiling the dataset which can be installed with the `openjdk-11-jdk` package in Ubuntu and then downloading Apktool 2.8.1 through their website. Figure 22 shows the whole process in the form of a diagram.



Figure 22: Extraction and decompilation process

APKtool Limitations

One of the limitations of using Apktool is that it can sometimes fail to decompile an APK file and thus cannot be analysed. Here are a few examples of errors that were discovered.

```
1 I: Using Apktool 2.8.1 on
   d4ad38f2e12655e2ea9ca6267ec088c61218be90db0b8593b2a8d67afbf462c1.apk
2 Exception in thread "main" brut.androlib.exceptions.AndrolibException:
   brut.directory.DirectoryException: java.util.zip.ZipException: Invalid
   CEN header (invalid zip64 extra data field size)
3
4 I: Using Apktool 2.8.1 on
   d65b7bbb0e96b15cbe707ca091c2d34ba2e159ffcf95bf3b4e9746074f55fea8.apk
5 Exception in thread "main" brut.androlib.exceptions.AndrolibException:
   brut.directory.DirectoryException: java.util.zip.ZipException: zip END
   header not found
6
```

Figure 23: APKtool error messages

The first error is due to an invalid CEN header, which is a documented issue within the Apktool code repository [63]. Unfortunately, there was no simple fix for this issue, so this error was accepted because it did not appear frequently. The second error message "zip END header not found" appears to be due to the APK file being corrupted [64], however, checking the SHA256 checksum shows that the file has not been modified when downloaded. As there is no solution to this error, it has to be accepted. Java 17 was also used to test the errors to ensure that the errors were not due by using a specific Java version, but the same error messages occurred. When these errors occur, Apktool still outputs the decompiled folder even though its empty. To bypass this problem, a cleanup must be performed to remove empty decompiled folders from the dataset as shown in Figure 24.


```

1 def cleanDecodedSamples(targetdir):
2     sampleCount = 0
3     fileCount = 0
4     for decodedSample in os.listdir(targetdir):
5         fileCount += 1
6         decodedSample = os.path.join(targetdir, decodedSample)
7         if not os.listdir(decodedSample):
8             os.rmdir(decodedSample)
9         else:
10            sampleCount += 1
11    print(f"[+] {fileCount} Files processed")
12    print(f"[+] {sampleCount} Files successfully decompiled")
13    print(f"[-] {fileCount - sampleCount} Invalid decompiled files removed")
14

```

Figure 24: Cleaning decoded samples

5.4 Extracting Features

5.4.1 Extracting Permissions

Extracting permissions requires going through each decompiled sample's AndroidManifest.xml file and then extracting the permissions from each <uses-permission> tag. Droidloader uses the following regular expressions to extract the permissions from the application. The regexr [65] website provides a description of how each of the special characters works.

```

1 <uses-permission.*?\/>
2 "(.*)"
3

```

Figure 25: Extracting permissions with Regex

The first expression finds all cases where the "<uses-permission>" start tag is written in the file followed by any characters, since each permission tag will have different text inside it. There are few cases in AndroidManifest.xml files where there are two <uses-permission> tags on one line, so the regex needs to accommodate for this edge case so that the permissions are treated separately. This is solved using this particular expression `*?` which is a reluctant quantifier that matches a pattern as few characters as possible. Figure 26 shows the differences between the use of a reluctant quantifier; permissions are treated as two separate elements in the list instead of one. Finally, the end of the expression checks if the tag is closed `\/>`. The second expression is more simple to implement, where it extracts the permission name that is in quotations.

```

1 >>> import re
2 >>> permissions = """<uses-permission android:name="android.permission.
    SEND_SMS"/> <uses-permission android:name="android.permission.SEND_SMS
    "/>"""
3 >>> print(re.findall(r'<uses-permission.*\/>', permissions))
4 ['<uses-permission android:name="android.permission.SEND_SMS"/> <uses-
    permission android:name="android.permission.SEND_SMS"/>']
5 >>> print(re.findall(r'<uses-permission.*?\/>', permissions))
6 ['<uses-permission android:name="android.permission.SEND_SMS"/>', '<uses-
    permission android:name="android.permission.SEND_SMS"/>']
7

```

Figure 26: Regex reluctant quantifier

Using pickle, the permissions are then saved as a Python object in the format of a list shown in Figure 27. The user can specify how many samples to extract from a given dataset with the default value being ten. The permission key values are stored as a set data type to avoid duplicate permissions being stored. The reason for storing it as a list is so that it can be iterated during data analysis.

```

1 [{ 'hash': '352070
    d3b4149fb4b28b030acfd60da0d143650eb643fd58ff12ddfc904f23c8', '
    permissions': {'android.permission.ACCESS_NETWORK_STATE', 'android.
    permission.READ_PHONE_STATE', ...}}]
2

```

Figure 27: Permissions data

5.4.2 Extracting Classes

Classes are more complex to extract compared to permissions as there are multiple files involved. There are two types of classes: the first are internal classes that are packaged within the malware, and the second are external classes which are classes the application uses but are not present in the package. The first thing to understand is the location of the smali code within a decompiled Android application. The smali code is stored in the smali folder, and sequential folders smali_classes starting from smali_classes1 depending on how large the malware sample is. The bigger the sample, the more smali_classes folders it contains. Within these folders, the smali code can be placed in subdirectories (Figure 28). Before extracting classes, the program needs to crawl for each smali file to generate a list of file paths to every smali file contained in the malware sample.

```

1 \smali
2     a.smali
3     \com
4         \android
5         ...
6     b.smali
7     ...
8 \smali_classes2
9 \smali_classes3
10 ...
11

```

Figure 28: Smali directory format

```

1 def getSmaliFolders(sourceCode):
2     return [file for file in sourceCode if re.search(r'smali_classes\d+|
3         smali', file)]
4 ...
5 smaliFolders = getSmaliFolders(sourceCode)
6 for folders in smaliFolders:
7     for root, _, files in os.walk(os.path.join(sourceCodePath, folders)):
8         ...

```

Figure 29: List of smali file paths

This function parses in the sourceCode, which is the directory containing the decompiled malware sample, and then returns a list containing the mali folder by searching for the relevant name using regex. After the list is returned, the os.walk function is called for each folder, which returns every file in a given folder including files in the subfolders.

Now that the program can iterate through each file, retrieving the classes that are packaged within the malicious file is relatively simple. Each smali file is essentially a class file with its class name defined at the top of the file as shown in Figure 30.

```

1 .class public final Landroid/support/v4/media/RatingCompat;
2

```

Figure 30: Smali class definition

The name is always declared at the end, so string manipulation can be performed to retrieve only the class name.

```

1 >>> className = ".class public final Landroid/support/v4/media/
   RatingCompat;"
2 >>> name = className.split()
3 >>> print(name)
4 ['.class', 'public', 'final', 'Landroid/support/v4/media/RatingCompat;']
5 >>> print(name[len(name)-1][1:-1])
6 android/support/v4/media/RatingCompat
7

```

Figure 31: Smali class name extraction

The split command splits each word separated by a whitespace into a list. The next command retrieves the last element of the list, which is the class name, and then removes the first and last character, as they are not relevant to the name. One of the limitations with this method is that the class names are obfuscated when decompiled, so even if the names are extracted, the results will be very different compared to looking at the original source code. After finding all the class names that are packaged within the malicious APK, the next step is to search for all the instances in each smali file where a class is invoked or referenced (const-class).

```

1 with open(smaliCode, "r") as f:
2     class = re.findall(r"invoke-virtual.*|invoke-super.*|invoke-direct.*|
   invoke-static.*|invoke-interface.*|const-class.*", f.read())
3

```

Figure 32: Finding invoke and reference calls

The | separator in regex is similar to the OR operator, where it will try to find any instances where it matches one of the invoke commands. This is then passed to the formatClass function which performs the following string manipulation, where it essentially only extracts the class names as shown in Figure 34.

```

1 def formatClass(item):
2     if "const-class" in item:
3         return item.split(" ")[2][:-1]
4     if "}," in item and ";->" in item:
5         return item.split("},")[1].split(";")[0].replace(" ", "")
6     else:
7         return ""
8

```

Figure 33: Extracting class names from invoke calls

```

1 ['invoke-direct {p0}, Ljava/lang/Object;-><init>()V', 'invoke-virtual {v0
   }, Ljava/lang/Class;->getName()Ljava/lang/String;']
2 ['java/lang/Object', 'java/lang/Class']
3

```

Figure 34: Extraction process output

With all the classes extracted, the last step is to format the classes and save them to disk as a Python object. Classes are categorised into three categories: internal, external, and all before being stored. The `internalClasses` variable is the result of getting all class names within the package as seen in Figure 31 and the `classes` variable is the result of extracting all invoke and reference instances (see Figure 34). Both of these variables are stored as a set data type to avoid repeated classes and to be able to perform set operations to create the mentioned categories, as shown in Figure 35.

```

1 classInfo = {
2     "hash": application,
3     "internal": internalClasses,
4     "external": classes - internalClasses,
5     "all": classes | internalClasses
6 }
7

```

Figure 35: Class name saved format

5.5 Processing Data

The permissions and classes stored as a pickle object when extracting features from an Android malware dataset need to be processed to produce results that can be compared. The first step is to compute the frequency distribution of the number of features observed in a dataset (Figure 36).

```

1 def getPermissionsFrequency(fileName, size):
2     permissionFrequency = Counter()
3     data = readData(fileName)
4     if data:
5         random.shuffle(data)
6         for application in data[:size]:
7             permissionFrequency += Counter(application['permissions'])
8     return permissionFrequency
9

```

Figure 36: Computing frequency distribution

`Counter` [66] is a Python Dictionary subclass from the Python collections module, where elements are stored as keys, and their counts are stored as dictionary values. This allows per-

missions to be added up for each sample in the database resulting in a dictionary containing the frequency for each unique permission observed. The same is applied to classes but there are separate frequency distributions for their three types (internal, external, all). The function also includes the size argument to allow for simple random sampling if only a desired amount of samples want to be taken from a dataset.

Formatting Data

Data are required to be formatted correctly before being plotted on the bar graph. There are too many permissions and classes to reasonably plot on the graph with more than 300 permissions [49] that can be defined and thousands of classes that can be defined for a single application; therefore, the aim is to retrieve the top n most common features for a dataset with their frequency, where n is the number of unique features to plot. The frequency would then be plotted against another dataset to see the differences in the usage of the specified features.

```

1 def processPermissions(file1, file2, categorySize, sampleSize=None):
2     dataset, dataset2, size = processSample(file1, file2, sampleSize)
3     Data = getPermissionsFrequency(file1, size)
4     Data2 = getPermissionsFrequency(file2, size)
5     category, frequency = formatData(Data, categorySize)
6     frequency2 = []
7     for permission in category:
8         frequency2.append(Data2[permission])
9     category = [permission.split(".")[0] if ".".join(permission.split("."))
10                 [0:2] == "android.permission" else permission for permission in
11                 category ]
12     processGraphData(category, "Permissions", categorySize, size,
13                     frequency, frequency2, dataset, dataset2)

```

Figure 37: Processing permission data

```

1 def formatData(data, categories):
2     data = data.most_common(categories)
3     return [item[0] for item in data], [item[1] for item in data]
4

```

Figure 38: Formatting data

In Figure 37, the frequency distributions of both datasets are retrieved from the getPermissionsFrequency function. The first distribution is passed into the formatData (Figure 38) function so that only the most common permissions are used when plotting the graph. The function also separates the permissions and frequency into two separate arrays so that the data are formatted correctly when plotted. Permissions are then filtered from the second distribution to include only

permissions that are the most common in the first distribution. As the majority of permissions are from Android, they are modified to include only the permission name if it starts with "android.permission". Any permissions that deviate from Android are given the full name to show that they are not part of Android specific permissions.

Processing Android API classes

One key data to plot is the Android API classes used for a dataset. The list of each package provided by Google [18] can be used to see which classes are contained in this package and therefore part of the Android API. Figure 39 shows a snippet of the `getClassesFrequency` function, which is similar to the `getPermissionsFrequency` but gets the data for all three types of classes. It works by filtering the classes through the `ValidateAPI` function, which returns all class names that start with any one of the names of the Android API package, showing that they are part of that package.

```
1 ...
2 for types in classTypes:
3     frequency2 = []
4     if types == "Android API":
5         types = 'all'
6         filteredData = {key: count for key, count in Data[types].items()
7 if validateAPI(key, androidAPI)}
8         category, frequency = formatData(Counter(filteredData),
9 categorySize)
```

Figure 39: Filtering Android API classes

```
1 def validateAPI(classes, AndroidAPI):
2     for package in AndroidAPI:
3         if classes.startswith(package):
4             return True
5     return False
```

Figure 40: Filtering Android API classes

Plotting Data

The Matplotlib module [67] can be used to create visualisation of data, such as the horizontal bar chart. The `processGraphData` function as shown in Figure 41 is created specifically to allow plotting for both classes and permissions, and also for different features that could be implemented in the future.

```

1 def processGraphData(category, featureType, categorySize, sampleSize,
2   frequency, frequency2, dataset, dataset2):
3     category = category[::-1]
4     frequency = frequency[::-1]
5     frequency2 = frequency2[::-1]
6     y = np.arange(len(category))
7     barWidth = 0.3
8     plt.barh(y - barWidth/2, frequency, barWidth, label=f"{dataset}")
9     plt.barh(y + barWidth/2, frequency2, barWidth, label=f"{dataset2}")
10
11     plt.xlabel('Frequency')
12     plt.ylabel(f'{featureType}')
13     plt.title(f'Top {categorySize} {featureType} from {dataset} Dataset
14     Compared to {dataset2} Dataset ({sampleSize} samples)')
15     plt.yticks(y, category)
16     plt.legend()
17     plt.show()

```

Figure 41: Plotting Graph

The order of the category and frequency lists needed to be inverted so that the most common feature appeared at the top of the bar graph instead of the bottom. The number of categories to display for the bar chart can also be changed to allow more features to be compared. The label and title of the graph are also dynamic, so that the plot description does not need to be manually changed for each feature.

5.6 Implementation of Testing Method

Androguard needs to be configured specifically to extract features to create tests to validate the permissions and class names extracted from a malicious sample. Specific classes and methods can be imported using the Androguard package. [68]. The analyseAPK function (Figure 42) shows how permissions, all classes, and external classes of an APK file are extracted by importing and using Androguard's classes and methods that perform all the processing. The full code for Androguard's and testing implementation can be seen in Appendix G.


```

1 from androguard.core.bytecodes import apk, dvm
2 import androguard.core.analysis.analysis as analysis
3
4 def analyseAPK(file):
5     apkFile = apk.APK(file)
6     dalvik = dvm.DalvikVMFormat(apkFile)
7     dx = analysis.Analysis(dalvik)
8     permissions = [permission for permission in apkFile.get_permissions()]
9     classes = [ dexClass.name[1:-1] for dexClass in list(dx.get_classes()) ]
10    externalClasses = [ dexClass.name[1:-1] for dexClass in list(dx.get_classes()) if dexClass.is_external() ]
11    return permissions, classes, externalClasses
12

```

Figure 42: Androguard Extracting Features

Figure 43 shows a snippet of the `getPermissionsAndClasses` function, which is used to retrieve and format features from a dataset using the Androguard package. It first calls `analyseAPK` to get all the features from Androguard, and then formats the permissions and classes to be the same format used in the Droidloader feature extraction process. Androguard cannot return internal classes but the internal class can be derived by excluding external classes from all classes, as shown in the set operation used (Figure 43). The hash data are included for both permissions and classes so that the program can match and compare the features extracted with Androguard to features extracted with Droidloader for a specific malware sample.

```

1 permissions = set()
2 classes = set()
3 externalClasses = set()
4 permissions, apkClass, apkExternalClass = analyseAPK(os.path.join(
5     targetDir, application,))
6 classes.update(apkClass)
7 classInfo = {
8     "hash": application[:-4],
9     "internal": classes - externalClasses,
10    "external": externalClasses,
11    "all": classes
12 }
13 permissionsList.append({"hash": application[:-4], "permissions": set(
14     permissions)})

```

Figure 43: `getPermissionsAndClasses` function snippet

The permissions extracted from Droidloader are then compared with Androguard's results to check if they are correct, as shown in Figure 44. The same concept applies to classes where each of the categories of classes are compared against Androguard.

```
1 def testGetPermissions():
2     permissions = getPermissions('datasets/TestDecoded', 10)
3     androguardPermissions = getPermissionsAndClasses('datasets/Test', 10)
4     [0]
5     successful = 0
6     for i in range(len(permissions)):
7         for j in range(len(androguardPermissions)):
8             if permissions[i]['hash'] == androguardPermissions[j]['hash']:
9                 successful += 1
10    print(f"[+] 10 samples tested")
11    print(f"[+] {successful} permissions from samples are correct
    compared to Androguard")
```

Figure 44: Validating permissions

5.7 Implementation of Benchmarking Method

To measure the performance of the tool, a benchmarking program was created that measures the time it takes for Droidloader to perform two actions: decompile a random dataset with Apktool and extract features. The combined time will be compared only with Androguard's time to extract features, since Androguard works with normal APK files. The code used for the automated benchmark can be seen in Appendix G where Figure 45 shows the key lines of the code. It first iterates through the selected values of the sampleSize array where each iteration generates a dataset in the datasets/Benchmark folder consisting of the specified number of malware samples. The program also ensures that the Benchmark folder is empty before generating the dataset by deleting and recreating the directory after each iteration. This dataset was then processed by Droidloader and Androguard, where its processing time is measured using the timeit module [69]. The times are then saved as a pickle object (Figure 46) so that every time is recorded for each benchmark test.

```

1 sampleSize = [10, 25, 50]
2 times = readData('data/Benchmark/times.pkl')
3 sampleDir = 'datasets/Benchmark'
4 decodedDir = 'datasets/BenchmarkDecoded'
5 ...
6 for size in sampleSize:
7     if os.path.exists(sampleDir):
8         shutil.rmtree(sampleDir)
9     os.mkdir(sampleDir)
10    generateSample(sampleDir, size)
11    execTimeDecompile = timeit.timeit(lambda: decompileTime(sampleDir,
12    decodedDir), number=1)
13    execTime = timeit.timeit(lambda: permissionsAndClassesTime(decodedDir
14    , size), number=1)
15    execTimeAndroguard = timeit.timeit(lambda:
16    permissionsAndClassesAndroguardTime(sampleDir, size), number=1)
17    times.append({"sampleSize": size, "decompileTime": execTimeDecompile,
18    "droidloader": execTime, "androguard": execTimeAndroguard})
19    writeData(times, "Benchmark", "times")

```

Figure 45: Measuring performance

```

1 [{ 'sampleSize': 10, 'decompileTime': 35.329267517000034, 'droidloader':
2.1299222259999624, 'androguard': 47.114663334}, { 'sampleSize': 25, '
decompileTime': 95.49785511700003, 'droidloader': 4.98925046100004, '
androguard': 97.97501675800004}, { 'sampleSize': 50, 'decompileTime':
135.027086436, 'droidloader': 8.33165734499994, 'androguard':
144.97242858799996}

```

Figure 46: Benchmarking data

6 Results and Evaluation

6.1 Generated Dataset Review

Two new datasets have been generated as a result of this tool. The first is MalwareBazaarRecent, which is the 1000 of the most recent samples taken from MalwareBazaar taken on August 22nd 2023. The second dataset is VirusShareRecent, which contains all Android malware that were filtered and downloaded from the VirusShare_00476.md5 [70] file. The CICAndMal2017 and CICMalDroid2020 dataset have been modified to exclude any Adware and the number of samples for the CICMalDroid2020 data has been reduced to make the size of the dataset similar to other datasets as well as storage limitations. Future references of the datasets in this section refer to their modified counterparts. Unfortunately, due to errors in the Apktool decompilation, some of the samples cannot be processed in the dataset including samples from previous datasets with Table 5 showing their success rate:

Dataset Name	Number of Malware Samples	Samples Successfully Decompiled	Success Rate (1 d.p.)
MalwareBazaarRecent	1000	842	84.2%
VirusShareRecent	774	633	81.8%
CICAndMal2017 (Modified)	322	322	100%
CICMalDroid2020 (Modified)	500	472	92.4%

Table 5: Dataset decompilation success rate

The average percentage of samples that have been successfully decompiled is 89.6% which is a sufficient percentage of samples that are processed. The number of malware samples for each dataset is small compared to datasets such as Drebin [33]. One of the issues of having a smaller sample size is that certain malware categories or families might be completely missed from the dataset. This could make the dataset represent less in the malware behaviour that was actually present from a certain time period. However, due to storage limitations and the CICAndMal2017 dataset having a small sample size, this cannot be avoided.

6.2 Testing and Benchmarking

Validating Extracted Permissions and Classes

Ten samples were randomly selected from all four datasets to create the test dataset where the features extracted from Droidloader are compared against Androguard. Figure 47 shows the results of the test for each file in which Droidloader correctly extracted the permissions and internal, external and all classes for each sample. This validates the feature extraction process.

Android Malware Sample MD5 Checksum	Permissions	Internal Classes	External Classes	All Classes	Testing Results:
80b6450d49dd03d0475b7f7fa23354c3					Features Correct
309fecc4cb7e80023ef83fd9ba75cfbb					Features Correct
02548535ff1cc285fddf699f2d77bcba					Features Correct
011889b7c5892e514b1c833d6bd4c476					Features Correct
d92f94c58c07e6c8ea65184b74e5c5c1					Features Correct
5a6ee791442ca33ddfc5fc1164b54557					Features Correct
ccb4b25c730e650ea05e6f7549c567c4					Features Correct
98bb2ab1112b5a792d97536828466cfb					Features Correct
5ef37fc05d43405e6b84ef50b4cfce7f					Features Correct
7e0ec2c932d9b6f7f8be7347aa9041a0					Features Correct

Figure 47: Feature extraction test results

Performance results

The time it took both tools to extract features from a dataset was recorded five times for each specific sample size. The average of those times is shown in Figure 48:

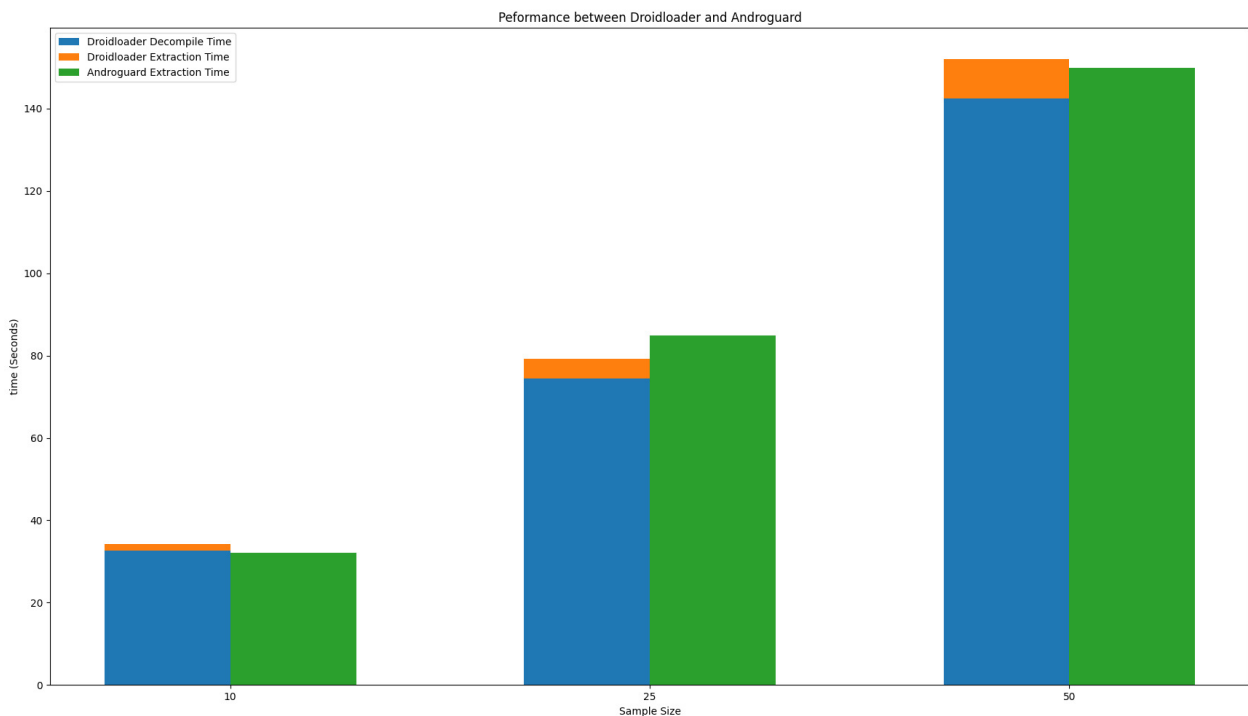


Figure 48: Performance between Droidloader and Androguard

From the graph, Droidloader and Androguard have similar results in performance. Decompiling took most of the time for Droidloader, but when it came to extracting features, it was significantly faster. Since decompiling a dataset is persistent, users can quickly extract features multiple times with Droidloader without having to decompile the dataset again. The graph also appears to show a linear relationship between the sample size and the time it takes to process the samples; therefore, working with thousands or more samples would not significantly affect the performance of the tool. However, one of the drawbacks of decompiling the dataset is the amount of storage it takes, where Table 6 shows the differences in size between a normal dataset and a decompiled dataset. While Droidloader can quickly extract features once the data are decompiled, disk storage had to be sacrificed.

Dataset Name	Size	Decompiled Size
MalwareBazaarRecent	9.5GB	56GB
VirusShareRecent	8.2GB	30GB
CICAndMal2017 (Modified)	895MB	5.6GB
CICMalDroid2020 (Modified)	1.4GB	5.9GB

Table 6: Dataset decompilation size comparison

6.3 Features Comparison

As there are four main datasets, twelve distinct bar charts could be made for each extracted feature; however, these many comparisons would not be necessary to provide sufficient results. To ensure that comparisons are standardised for each feature, the datasets used for each plot will be the same. There would be one plot between the newly generated dataset (MalwareBazaarRecent) and the older dataset (CICAndMal2017), as well as a plot between the newly generated datasets (MalwareBazaarRecent and VirusShareRecent) and a plot between the older datasets (CICAndMal2017 and CICMalDroid2020). Any significant differences in frequency for unique features will be discussed in the section and will provide insight into why the differences occurred. The same number of samples are taken from both datasets so that the frequency distribution is not weighted for one dataset or the other.

6.3.1 Permission Data Comparison

Figure 49 shows the top twenty most common permissions used in the MalwareBazaarRecent dataset compared to the CICAndMal2017 dataset. The additional plots for Android permissions can be seen in Appendix A, which is also analysed in this section.

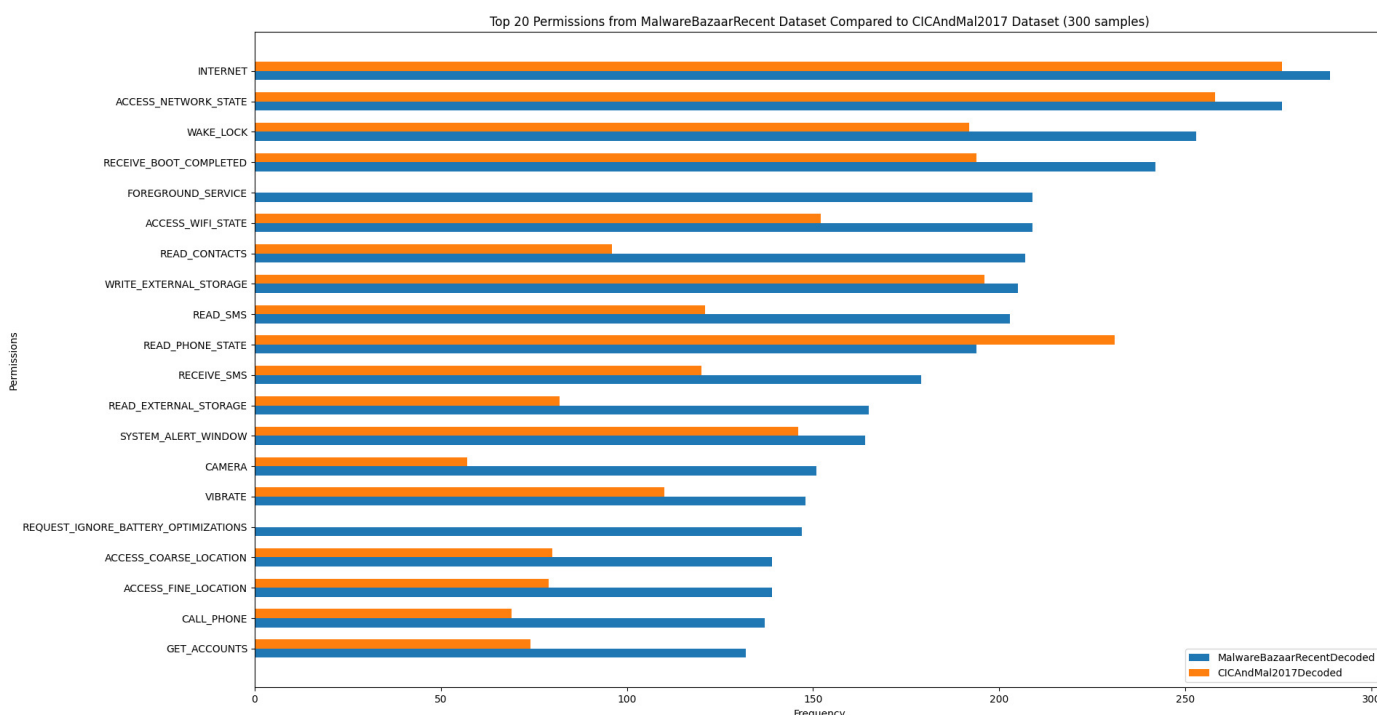


Figure 49: Permission frequency between MalwareBazaarRecent and CICAndMal2017

First of all, there are two permissions that are not included in the CICAndMal2017 dataset which are `FOREGROUND_SERVICE` and `REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`. The first permission, `FOREGROUND_SERVICE` was added in API level 28 [49] which correlates with Android 9. This version of Android was released publicly in August 2018, so the permission would not be present in malware that was released before the version. Foreground services can be exploited to execute tasks from the background context [71]. Exploiting these flaws allows an application to use the device's resources, even when it is closed. The second permission does not look malicious at first as it is used to simply ignore battery optimisation. However, this permission can be exploited to prevent Doze [72] which restricts certain access and actions for an application when the device is on sleep to preserve battery. Malware authors use this permission so that malware, particularly remote access trojans, can constantly connect back to their command and control servers. This permission was introduced in Android 6, released in September 2015, so there is no concrete reason why none of the samples have this permission set in the older dataset. Some of the samples collected in the CICAndMal2017 dataset have evidence of C&C activity [73] so the reason is not due to biased malware types.

The frequency distribution of permission is also different between MalwareBazaarRecent and VirusShareRecent (see Appendix A) despite the malware samples being taken from the same time period. Both datasets do not follow the same pattern in frequency distribution as expected, with a few notable permissions that have major differences in their frequency. The first are permissions related to SMS messages and contacts reconnaissance (`READ_SMS`, `READ_CONTACT`,

RECIEVE_SMS) which the majority of samples from VirusShareRecent dataset do not have this permission set. This could be due to malware sample bias, where the samples taken from VirusShare may contain different malware types such as adware that does not need access to contact information. Other permissions include QUERY_ALL_PACKAGES, which allows the malware to query any normal application, and USE_FULL_SCREEN_INTENT, used to display a full-screen notification. Displaying a notification can trick the user into potentially clicking it to grant the malware more access to the device. These permissions are not as intrusive as SMS and contact permissions, so their usage depends on the specific goal that the malware is trying to achieve. All these permissions mentioned were also available to use when the malware was written from both datasets, so malware authors had the option to include these permissions to maximise their malware capability.

Lastly, the permission distribution between the CICAndMal2017 and CICMalDroid2020 dataset roughly follows the same pattern, with no particular permissions standing out from the distribution. In general, there are significant changes in the frequency distribution of permissions from older malware samples to newer malware samples. This suggests behavioural changes in malware, making older datasets not as effective as training data. However, there is also evidence of behavioural changes even when malware is compared between datasets from the same time period. It suggests that generating new datasets is not optimal for Android malware detection based on machine learning; malware behaviours can be vastly different depending on their type, so further processing should be used where malware is categorised depending on their capability. The categorised data should then be used to detect specific malware types, for example, creating a detection model to detect unknown banking malware should only be trained with banking malware.

6.3.2 Classes Data Comparison

Producing meaningful results from comparing the frequency of classes used for each dataset is more difficult than comparing permissions, as the methods used for each class are not known in the data; it is difficult to know the full intent of why the author used a specific class or Android API. When it came to producing graphs for the classes in the external and all category, the bar charts produced similar results to classes that were part of the Android API package (see Appendix B). Therefore, only the frequency of classes that are part of the Android API package will be analysed. As a result, a new category of classes will also be plotted, which are all classes that are not part of the Android API package. Overall, this section reviews three distinct categories of classes: internal, part of the Android API package, and not part of the Android API package.

Internal Classes

Figure 50 shows the top twenty classes that were packaged within the malware samples of the MalwareBazaarRecent dataset compared to CICAndMal2017. Additional plots for internal classes can be found in Appendix C.

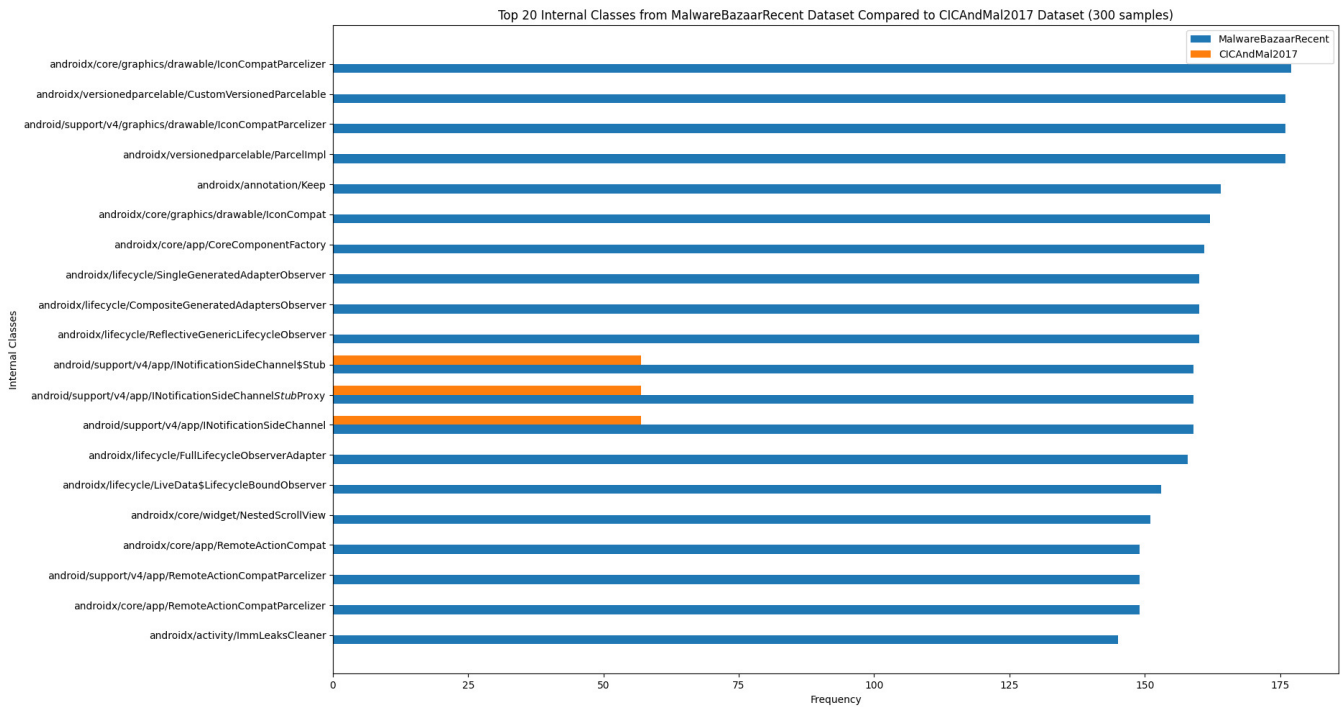


Figure 50: Internal classes frequency between MalwareBazaarRecent and CICAndMal2017

Looking at the internal classes, the majority of samples in the newer dataset use the androidx package, which is not present in the older dataset. This is because Androidx is a major improvement to the original Android support library that is no longer maintained [74]. The graph shows the older dataset's samples using the outdated support library. This first stable release of Androidx was in September 2018 [75] which dates after the release of malware in the CICAndMal2017 dataset. The actual package provides backward compatibility within the android application, so it is not malicious. However, malware authors could use this to increase the number of Android versions that are compatible with the malware so that more devices can be targeted.

There is an anomaly with the MalwareBazaarRecent and VirusShareRecent datasets (see Appendix C), where most of the malware in the VirusShareRecent dataset do not include the Androidx package. This indicates that the VirusShareRecent malware is outdated, and so the malware is not as recent as expected. This could have been one of the reasons for the unexpected results when comparing both of the mentioned datasets permission frequency distribution, as malware from both datasets was created at the same time. However, this does not completely rule out the possibility of malware type biases in the generated datasets.

Android API Classes

Figure 51 shows the top twenty classes that are part of the Android API packaged from the MalwareBazaarRecent dataset that are compared to the CICAndMal2017 datasets. Additional plots for Android API classes can be found in Appendix D.

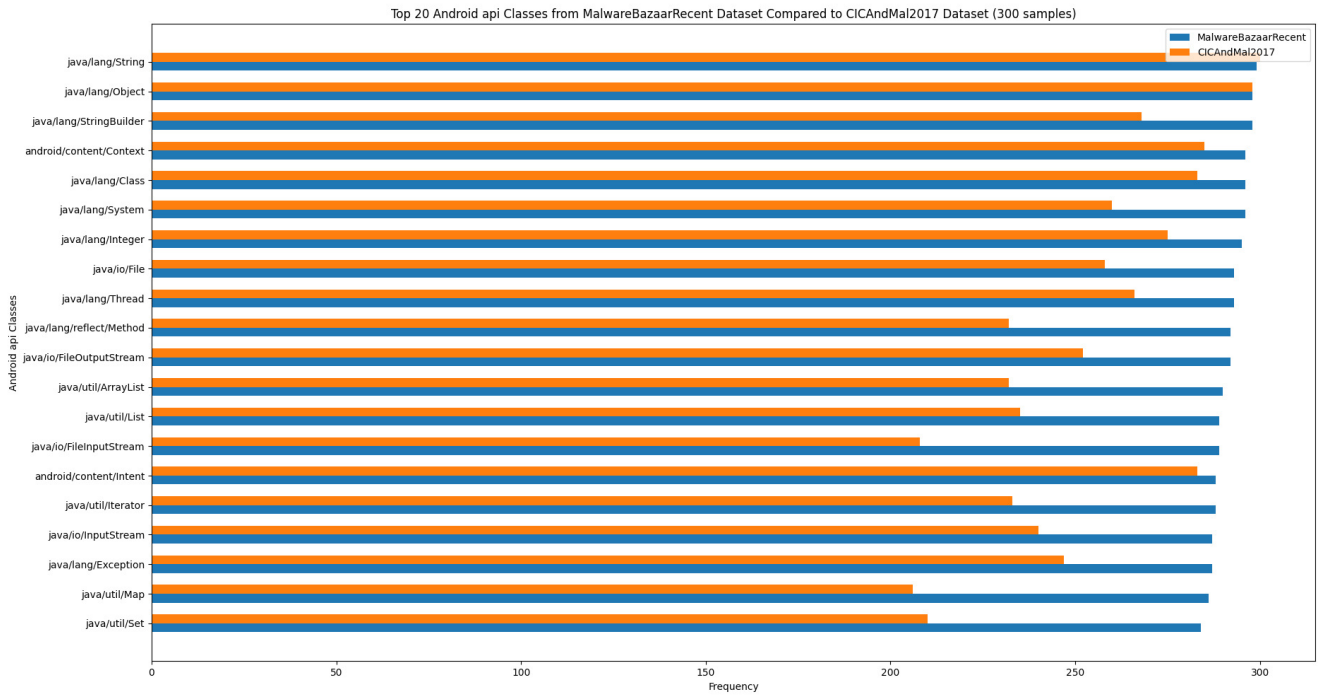


Figure 51: Android frequency between MalwareBazaarRecent and CICAndMal2017

Viewing all the plots for the frequency distributions of Android API classes (see Appendix D), show no real differences in the use of Android API classes between datasets other than the MalwareBazaarRecent and VirusShareRecent dataset. VirusShareRecent has numerous API classes that appear only more than half the time compared to MalwareBazaarRecent. A few notable API classes include `Android/content/BroadcastReceiver` and `android/net/URI`. The first API class allows malicious applications to trigger certain actions based on which broadcast intent was received [76]. Broadcast intents are system-wide intents delivered to each app when an event happens, such as unlocking the screen. The second API class can be used to help build and manipulate URI references [77], which could be used to send specific http requests to C&C servers.

Although the other datasets differed less in Android API class usage compared to MalwareBazaarRecent and VirusShareRecent, there are few instances of API Classes having a notable difference in usage between datasets. These include: `java/io/FileInputStream` and `android/text/TextUtils` between MalwareBazaarRecent and the CICAndMal2017 dataset as well as `android/content/SharedPreferences` and `android/content/SharedPreferences$Editor` between CICAndMal2017 and CICMalDroid2020. `FileInputStream` obtains input bytes from a file in a file system [78] so it could be used to read files. `TextUtils` is used to perform operation on a `String` object, making it useful for decoding any obfuscated strings within the malware. Lastly, `SharedPreferences` [79] is an interface that allows accessing and modifying preference data that could be used to retrieve stored data. Altogether, comparing the Android API classes has provided some highlights of API classes malware authors can use to maximise the impact of the

malware. However, the differences in Android API usage were not significant enough compared to the permissions and internal classes frequency distributions to justify behavioural changes in Android API usage over time. Not every malware is going to be the same for each dataset so some differences in frequency are expected.

Non Android API Classes

Figure 52 shows the top twenty classes that are not part of the Android API package from the MalwareBazaarRecent dataset compared to the CICAndMal2017 dataset. Additional plots for Non Android API classes can be found in Appendix E.

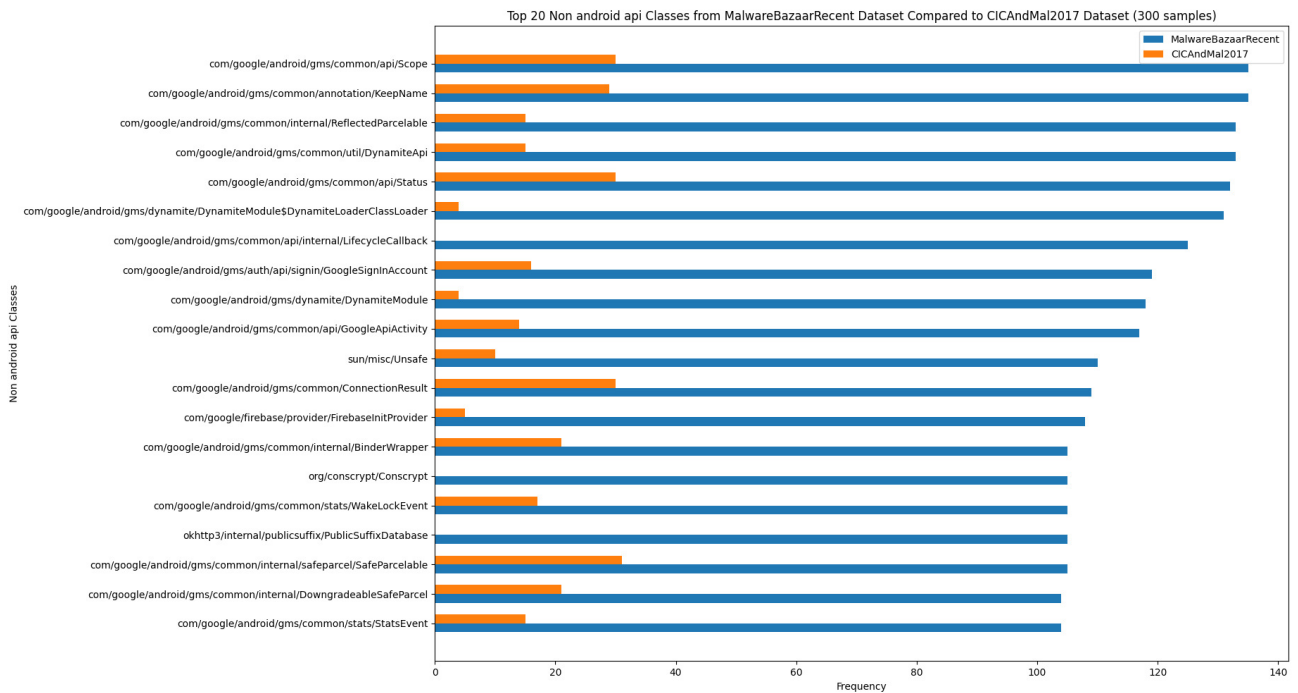


Figure 52: Non Android API classes frequency between MalwareBazaarRecent and CICAndMal2017

Both the MalwareBazaarRecent and CICAndMal2017 plots, and the MalwareBazaarRecent and VirusShareRecent plots show no real pattern in their respective frequency distribution of non-API classes (see Appendix E). All the differences in non API classes usage in the mentioned dataset are very different with no specific classes having consistent usage. Some classes that stand out are those that belong to the okhttp3 and google package. The okhttp3 package [80] is an HTTP client that can accept requests and produce responses, such as posting data to a service. HTTP activity could suggest that the malware is trying to exfiltrate data to a command and control server, by using the HTTP protocol to mask the activity [81]. There were no suspicious google package classes other than Google Firebase as the Firebase messaging service has been used by threat actors in the past to download malicious payloads [82]. Overall, the variable differences in the

use of these classes signify behavioural changes in the specified datasets. The CICAndMal2017 and CICMalDroid2020 datasets show a more consistent usage in non API classes, but that is to be expected as the malware are from the same time period; malware in these datasets primarily uses Apache to perform http services, and as mentioned above, these services can allow malware to communicate to external servers such as C&C servers.

7 Conclusions

In conclusion, the project reviews the existing research in ML-based Android malware detection and recognises the main problem where most of the frameworks use outdated Android malware dataset to train their framework. After reviewing the evolution and current threats of Android malware, it is apparent that its behaviour and capabilities change throughout the years. As a result, detection rates were not as effective when testing against new malware due to significant changes in their behaviours.

To solve this problem, the Droidloader tool was created, which is a command-line interface tool that automates the generation and analysis of new datasets. It was decided that the malware sources would be from MalwareBazaar and VirusShare as they both provide an API to allow the tool to interact directly with both sources. The MalwareBazaar API allows the retrieval of Android malware and has a strict process for uploading malware samples to the website, so the chances of false positives in the generated dataset were negligible. However, the MalwareBazaar API only allowed a limited number of samples that could be downloaded. A different approach had to be designed for VirusShare as Android malware cannot be queried directly from the VirusShare API, as well as their database potentially containing false positive malware. This was done by using the VirusTotal API so that it could filter for VirusShare Android malware and remove false positives. However, the request limit for both VirusTotal and VirusShare meant that it would take a few days to download a sufficient amount of Android malware samples. Despite this, there is no hard limit on the number of malware samples that could be obtained. As a result of the tool, two new datasets, MalwareBazaarRecent and VirusShareRecent were generated. By using APIs, Droidloader can dynamically generate new datasets where if the tool was used in the future, it will generate a new dataset with malware created in that future time period.

After generating the datasets, Droidloader had to extract features from the dataset so that comparisons could be made between two separate datasets. By covering various static and dynamic techniques used to extract features from previous academic research, it was decided that the tool will perform only static analysis, as it is a more suitable technique when processing large amounts of malware samples. The features chosen to extract were the permissions and classes defined in each malware sample. More features could have been analysed, but the scope of the deliverable had to be reduced to ensure it was complete before the deadline. However, there was a problem when processing the dataset as Apktool was unsuccessful in decompiling some of the malware samples. This proved not to be a major issue with Apktool successfully decompiling 89.6% of the samples from the chosen datasets.

When the features were extracted, they could then be plotted in a double bar chart for further analysis by comparing the frequency distribution of the features used in one dataset with the other. Two datasets, CICAndMal2017 and CICMalDroid2020, created from previous academic research were chosen to be compared with the generated dataset. Other datasets that are more widely used, such as Drebin [33] would have been preferred, but access to this dataset was not granted during the project period.

To ensure that Droidloader correctly extracted the features, Androguard was used to test the output of the extracted classes and permissions from ten malware samples. Ten samples were used to ensure that the extraction process could work for any sample and the test verified that Droidloader could correctly extract all the features of the malware samples.

The performance of the tool was similar to that of Androguard in extracting features. But once the datasets were decompiled, Droidloader performs faster, which justifies the reason for designing and implementing individual methods to extract features. A dataset is only required to be decompiled once before numerous feature extraction can be performed. But, as a consequence of the fast performance, decompiled datasets take up a large chunk of disk space, so there needs to be some optimisation to remove any files not relevant in the analysis from decompiled files.

After Droidloader could extract features, the comparisons of datasets could be made through plotting double bar charts. The plots showed that the frequency distribution of permissions for both the outdated dataset and the new dataset was significant enough to indicate behaviour changes. This was due to some of the permissions being added in later versions of Android, which meant that the permissions were not available for malware that was part of an older dataset. However, the analysis also showed unexpected results where it indicated behavioural changes within the two generated, including malware from the same time period. As a consequence, only generating a new dataset might not be sufficient for training data, and further processing, such as categorising the malware type, should be considered. When it came to comparing the classes, the plots showing the frequency distribution of the internal and non Android API classes were very different indicating behaviour changes within Android malware. The reasons for these changes could be due to newer and better classes. However, the analysis process revealed that the VirusShareRecent database could be outdated, as it contained old internal classes. Analysis of the frequency of Android API classes between datasets also showed no promising results, as usage was generally consistent for different time periods of malware.

By designing Droidloader to be a command-line tool, users are given the flexibility to download different sizes of dataset, choose a specific number of samples to analyse, and change the number of categories to display when plotting graphs. The code does not need to change to allow for specific commands in the tool. However, the documentation for the tool is not present and there was no unit testing performed, so the behaviour of the tool could drastically change or crash if a user enters an unexpected input.

Overall, this project has achieved all the measurable goals specified which led to the creation of a newly generated Android malware dataset, where their behaviour was compared with an older Android malware dataset. The results indicated behavioural changes reinforcing the problem of using outdated datasets, but it also introduced a whole new problem of malware type biases within generated datasets.

8 Future Work

There are various avenues through which this project can be further developed for future work. A few ideas include making further improvements to the tool itself, or expanding on the initial problem outlined in this project, which could lead to original research for a PhD thesis. This section covers a few key areas for the future work of this project.

Reducing Malware Type Biases in Datasets

One of the problems discovered in this project was the potential malware-type biases in the generated datasets. This is an issue because the dataset could be biased towards specific malware types and exclude certain categories within the dataset as there are no processes to categorise the malware. One of the problems when it comes to malware type categorisation is malware labelling. Different anti-virus vendors will label a virus differently even if it is in the same family or type. Tools such as AVClass [83] try to solve this problem by labelling malware behaviours, types, and file properties based on a certain class based on multiple anti-virus labelling results. The most common type that is labelled for a malware sample can then be used to categorise the malware. There might also be other ways to categorise malware that do not rely on anti-virus labelling, such as producing a unique methodology that categorises the malware by type if exhibits a certain behaviour when executed. There was also an issue with generated datasets that contained outdated malware, so a filtering system needs to be implemented to only accept malware from a certain date. By creating a solution that also categorises malware after it is generated, it could provide some novel contribution that can lead to a PhD.

Improving Droidloader

The Droidloader tool was created to facilitate the project and, although it achieves the objectives set in the project, there are few areas in this tool that can be further explored. The commands and options used for the interface were mainly decided by personal preferences, which might not be clear to a new user. So the first step is to improve the user experience, as currently the tool has no supporting documentation; users have to go through multiple commands to generate the decoded database, which might not be easy to perform for a new user. Creating documentation for the tool would be useful so that users know what each command does and the main use case of the tool. Improvement to the interface could include clearer arguments or options, so instead of `-s` for specifying the size, use `--size`. Once the changes have been made with the interface, numerous users who have never experienced the tool can test to see if the interface is easy to use by attempting to successfully extract and plot features from a specified database. If many users have failed the test, then it would be clear that the tool needs my work in its user experience. Another option would be to create a graphical user interface where users can drag the dataset folder into the program and it will do all the processing in the background.

Another improvement to consider is making the tool open source so that external developers can contribute to the code as well as make provide issues of requests for certain features. Although

the code styling is consistent with Droidloader, it does not conform to PEP 8 [84] which provides standard guidelines and practise on how to write Python code. Droidloader would have to conform to this standard before its released to the open source community so that the code styling is consistent since every developer would know and use this standard even if they have a certain styling preference. Lastly, some optimisation changes should be implemented in Droidloader, for example, reducing the file size of decompiled datasets as they are large compared to the unmodified datasets. One of the ways that this could be done is by removing folders and files that are not relevant for feature extraction within the malware after decompilation.

Extracting More Features

Only static analysis was used for the comparison of the dataset and only a limited number of features were extracted. While the classes were successfully extracted, the methods of the classes were not obtained, which limited the analysis section, as there was no context behind why each class was initialised within the malware. Another idea would be to use dynamic analysis methods, where the tool automates the execution of each application so that it can generate data for analysis. This can be done by using adb [26] to load the application onto a physical or emulated mobile device, and then using monkey runner [23] to simulate actions. Data such as network traffic and system calls can then be extracted, which could provide some useful behavioural indicators for given malware samples. With more features extracted, a more thorough analysis could be considered to further validate behavioural changes within malware.

Implementing Machine Learning

Lastly, machine learning could be used to quantify behaviour changes in malware from outdated datasets and generated sets. Two separate machine learning models for Android malware detection could be created, where one is trained on the generated datasets and the other on outdated datasets. The models would then be tested on unknown malware to see if they can correctly detect it. The detection accuracy could then be compared between both models to see if using generated datasets would improve ML-based Android malware detection. If improvement is observed, then there is evidence of behavioural changes, since the new model could adapt to the new changes in malware.

9 Project Reflection

I initially found the project challenging in the beginning with limited knowledge about how the Android ecosystem worked on a technical level, including how Android malware are packaged, deployed, and executed on a system. It was definitely a daunting experience starting a project with little prior knowledge in the project background area. I also found that Android does not have as much documentation and research compared to other systems, such as Windows, making it more difficult to fully understand the key concepts. However, after dedicating the first few weeks of learning Android and its malware by playing around with APK files and reading tutorials and books, I felt confident in the foundational knowledge I had about Android and its relevant malware to start the dissertation project.

One of the main issues I faced was coming up with a suitable and novel project idea. This project was initially going to analyse the vulnerabilities of Android mobile applications by writing mitigation techniques to prevent the vulnerabilities; however, there was limited academic literature and so the project would have just been a paper detailing past vulnerabilities without insightful research. But then I started reading academic papers on Android malware, particularly in ML-based Android malware detection where I discovered the key problem which was that most of the solutions used outdated datasets. This problem then provided a base for my project, which I am glad I chose, as it meant that I could critically analyse the literature and provide insightful results to solve the problem.

One of the key strengths I had going into this project was having a strong programming background, especially in Python. Therefore, I wanted to make this project programming-based, which is why I decided to write the Droidloader tool. The experience greatly helped as I could immediately write the code for Droidloader instead of spending time learning the language. The experience also made sure that I took a modular approach in programming, where separate files and functions performed a particular task. This approach allowed me to make small tweaks to the code towards the end without too much friction, and also avoid redundant code. Python's third-party libraries also saved a lot of time, as I did not have to build some of the features by scratch, such as using the `argparse` module to provide command-line options for the user. It was definitely daunting to write the tool at first, as there were some concepts with which I did not have much experience, such as sending and retrieving API requests and file manipulation. But with the help of documentation forums, these concepts were manageable to implement. I also had little experience in regex, but the expressions required to write were not too complex, so I did not have to learn all the syntax in regex.

The major limiting factor in this project was definitely time. There were some ideas that I wanted to implement in the project, for instance, dynamic analysis to extract features. However, there were external factors outside my control working on the project that meant that I could not fully focus on the project. Therefore, I made the decision to reduce the scope of the tool so that I had a completed product for the project. Other factors also include a lack of knowledge in other

areas, as I wanted to use machine learning in the project; however, I never had the fundamental knowledge of how machine learning works before starting the project, so I decided against it.

Whilst doing this project, I have massively improved my research skills by skimming through academic papers to determine if it is worth reading and utilising their references to find even more papers. My time management skills have improved significantly, as I delegated enough time to writing the project so that the work was manageable. I also have greater reliance when it comes to learning new concepts, as I do not get too overwhelmed when I am struggling, but instead try to take a more calm approach. Overall, I found this project to be an enjoyable experience from start to finish, especially when it came to programming the tool. I have learnt many skills that I can put into practise in the future and I hope one day that I can release my tool to the public so that my tool can aid researchers in the future.

References

- [1] Statcounter. *Desktop vs Mobile vs Tablet Market Share Worldwide*. en. URL: <https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/> (visited on 30/08/2023).
- [2] *Mobile Operating System Market Share Worldwide*. en. URL: <https://gs.statcounter.com/os-market-share/mobile/worldwide> (visited on 28/06/2023).
- [3] Ali Muzaffar et al. ‘An in-depth review of machine learning based Android malware detection’. en. In: *Computers & Security* 121 (Oct. 2022), p. 102833. ISSN: 0167-4048. DOI: 10.1016/j.cose.2022.102833. URL: <https://www.sciencedirect.com/science/article/pii/S0167404822002279> (visited on 25/07/2023).
- [4] ThreatFabric. *2022 mobile threat landscape update*. en. URL: <https://www.threatfabric.com/blogs/h1-2022-mobile-threat-landscape> (visited on 30/08/2023).
- [5] Yajin Zhou and Xuxian Jiang. ‘Dissecting Android Malware: Characterization and Evolution’. In: *2012 IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2012, pp. 95–109. DOI: 10.1109/SP.2012.16.
- [6] Ashawa Moses and Sarah Morris. ‘Analysis of Mobile Malware: A Systematic Review of Evolution and Infection Strategies’. en. In: *Journal of Information Security and Cybercrimes Research* 4.2 (Dec. 2021). Number: 2, pp. 103–131. ISSN: 1658-7790. DOI: 10.26735/KRVI8434. URL: <https://journals.naass.edu.sa/index.php/JISCR/article/view/1578> (visited on 22/06/2023).
- [7] KrebsonSecurity. *How Malicious Android Apps Slip Into Disguise – Krebs on Security*. en-US. Aug. 2023. URL: <https://krebsonsecurity.com/2023/08/how-malicious-android-apps-slip-into-disguise/> (visited on 19/08/2023).
- [8] *ANDROIDOS_DROIDSMS.A - Threat Encyclopedia*. URL: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/ANDROIDOS_DROIDSMS.A (visited on 15/08/2023).
- [9] Kimberly Tam et al. ‘The Evolution of Android Malware and Android Analysis Techniques’. en. In: *ACM Computing Surveys* 49.4 (Dec. 2017), pp. 1–41. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3017427. URL: <https://dl.acm.org/doi/10.1145/3017427> (visited on 22/06/2023).
- [10] Symantec. ‘Mobile Adware and Malware Analysis’. en. In: ().
- [11] Kaspersky. *Mobile cyberthreat report for 2022*. en-US. Feb. 2023. URL: <https://securelist.com/mobile-threat-report-2022/108844/> (visited on 18/08/2023).
- [12] Kaspersky. *Google Play threat market: overview of dark web offers*. en-US. Apr. 2023. URL: <https://securelist.com/google-play-threats-on-the-dark-web/109452/> (visited on 19/08/2023).

- [13] ThreatFabric. ‘The State of Android (Banking) Malware’. en. In: (Feb. 2022). URL: https://www.threatfabric.com/hubfs/ThreatFabric_Generic_Report-The%20State%20of%20Android%20Banking%20Malware%202022.pdf.
- [14] Meaghan Yuen. *State of mobile banking in 2022: top apps, features, statistics and market trends*. en. URL: <https://www.insiderintelligence.com/insights/mobile-banking-market-trends/> (visited on 28/08/2023).
- [15] Fauzia Idrees et al. ‘AndroPIn: Correlating Android permissions and intents for malware detection’. In: *2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. Oct. 2017, pp. 394–399. DOI: 10.1109/IEMCON.2017.8117152.
- [16] Benjamin Aruwa Gyunka, Aro Taye Oladele and Ojeniyi Adegoke. ‘Adaptive Android APKs Reverse Engineering for Features Processing in Machine Learning Malware Detection’. en. In: *International Journal of Data Science* 4.1 (May 2023). Number: 1, pp. 10–25. ISSN: 2722-2039. DOI: 10.18517/ijods.4.1.10-25.2023. URL: <http://ijods.org/index.php/ds/article/view/66> (visited on 01/08/2023).
- [17] Yajin Zhou et al. ‘Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets’. In: *Proceedings of the 19th Network and Distributed System Security Symposium NDSS 2012* (Jan. 2012).
- [18] Google. *Package Index*. en. URL: <https://developer.android.com/reference/packages> (visited on 28/08/2023).
- [19] Moutaz Alazab et al. ‘Intelligent mobile malware detection using permission requests and API calls’. In: *Future Generation Computer Systems* 107 (June 2020), pp. 509–521. ISSN: 0167-739X. DOI: 10.1016/j.future.2020.02.002. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X19321223> (visited on 28/08/2023).
- [20] Eduardo Blázquez and Juan Tapiador. ‘Kunai: A static analysis framework for Android apps’. en. In: *SoftwareX* 22 (May 2023), p. 101370. ISSN: 23527110. DOI: 10.1016/j.softx.2023.101370. URL: <https://linkinghub.elsevier.com/retrieve/pii/S2352711023000663> (visited on 01/07/2023).
- [21] skylot. *skylot/jadx*. original-date: 2013-03-18T17:08:21Z. Aug. 2023. URL: <https://github.com/skylot/jadx> (visited on 28/08/2023).
- [22] Kimberly Tam et al. ‘CopperDroid: Automatic Reconstruction of Android Malware Behaviors’. en. In: *Proceedings 2015 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2015. ISBN: 978-1-891562-38-9. DOI: 10.14722/ndss.2015.23145. URL: <https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/copperdroid-automatic-reconstruction-android-malware-behaviors/> (visited on 28/08/2023).

- [23] Google. *monkeyrunner* | *Android Studio* | *Android Developers*. en. URL: <https://developer.android.com/studio/test/monkeyrunner> (visited on 29/08/2023).
- [24] Xinrun Zhang et al. ‘An Early Detection of Android Malware Using System Calls based Machine Learning Model’. en. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. Vienna Austria: ACM, Aug. 2022, pp. 1–9. ISBN: 978-1-4503-9670-7. DOI: 10.1145/3538969.3544413. URL: <https://dl.acm.org/doi/10.1145/3538969.3544413> (visited on 28/08/2023).
- [25] Genymotion. *Android Emulator on the Cloud and cross-platform - Genymotion*. en-US. URL: <https://www.genymotion.com/> (visited on 28/08/2023).
- [26] Google. *Android Debug Bridge (adb)* | *Android Studio*. en. URL: <https://developer.android.com/tools/adb> (visited on 28/08/2023).
- [27] Yung-Ching Shyong, Tzung-Han Jeng and Yi-Ming Chen. ‘Combining Static Permissions and Dynamic Packet Analysis to Improve Android Malware Detection’. In: *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*. June 2020, pp. 75–81. DOI: 10.1109/ICCCI49374.2020.9145994.
- [28] Jyoti Malik and Rishabh Kaushal. ‘CREDROID: Android malware detection by network traffic analysis’. en. In: *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*. Paderborn Germany: ACM, July 2016, pp. 28–36. ISBN: 978-1-4503-4346-6. DOI: 10.1145/2940343.2940348. URL: <https://dl.acm.org/doi/10.1145/2940343.2940348> (visited on 29/08/2023).
- [29] Vikas Sihag, Manu Vardhan and Pradeep Singh. ‘A survey of android application and malware hardening’. In: *Computer Science Review* 39 (Feb. 2021), p. 100365. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2021.100365. URL: <https://www.sciencedirect.com/science/article/pii/S1574013721000058> (visited on 15/08/2023).
- [30] Mijoo Kim et al. ‘A study on behavior-based mobile malware analysis system against evasion techniques’. In: *2016 International Conference on Information Networking (ICOIN)*. Jan. 2016, pp. 455–457. DOI: 10.1109/ICOIN.2016.7427158.
- [31] Hugo Gascon et al. ‘Structural detection of android malware using embedded call graphs’. en. In: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. Berlin Germany: ACM, Nov. 2013, pp. 45–54. ISBN: 978-1-4503-2488-5. DOI: 10.1145/2517312.2517315. URL: <https://dl.acm.org/doi/10.1145/2517312.2517315> (visited on 29/08/2023).
- [32] Wei Wang et al. ‘Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection’. In: *IEEE Transactions on Information Forensics and Security* 9.11 (Nov. 2014). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 1869–1882. ISSN: 1556-6021. DOI: 10.1109/TIFS.2014.2353996.

- [33] Daniel Arp et al. 'Drebin: Effective and Explainable Detection of Android Malware in Your Pocket'. en. In: *Proceedings 2014 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2014. ISBN: 978-1-891562-35-8. DOI: 10.14722/ndss.2014.23247. URL: <https://www.ndss-symposium.org/ndss2014/programme/drebin-effective-and-explainable-detection-android-malware-your-pocket/> (visited on 29/08/2023).
- [34] Hyoil Han et al. 'Enhanced Android Malware Detection: An SVM-Based Machine Learning Approach'. In: *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*. ISSN: 2375-9356. Feb. 2020, pp. 75–81. DOI: 10.1109/BigComp48618.2020.00–96.
- [35] Ming Fan et al. 'Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis'. In: *IEEE Transactions on Information Forensics and Security* 13.8 (Aug. 2018). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 1890–1905. ISSN: 1556-6021. DOI: 10.1109/TIFS.2018.2806891.
- [36] VirusShare. *VirusShare*. URL: <https://virusshare.com/> (visited on 28/08/2023).
- [37] Kai Chen et al. 'Finding unknown malice in 10 seconds: mass vetting for new threats at the Google-play scale'. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. USA: USENIX Association, Aug. 2015, pp. 659–674. ISBN: 978-1-931971-23-2. (Visited on 29/08/2023).
- [38] VirusTotal. *VirusTotal - Home*. URL: <https://www.virustotal.com/gui/home/upload> (visited on 20/08/2023).
- [39] R. Jordaney et al. *Transcend: Detecting Concept Drift in Malware Classification Models*. eng. Proceedings paper. Conference Name: 26th USENIX Security Symposium Meeting Name: 26th USENIX Security Symposium Num Pages: 18 Pages: 625-642 Place: Vancouver, Canada Publisher: USENIX Association. Jan. 2017. URL: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney> (visited on 29/08/2023).
- [40] Arash Habibi Lashkari et al. 'Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification'. In: *2018 International Carnahan Conference on Security Technology (ICCST)*. ISSN: 2153-0742. Oct. 2018, pp. 1–7. DOI: 10.1109/CCST.2018.8585560.
- [41] Mila. *Take a sample, leave a sample. Mobile malware mini-dump - July 8 Update*. July 2011. URL: <https://contagiodump.blogspot.com/2011/03/take-sample-leave-sample-mobile-malware.html> (visited on 06/09/2023).
- [42] Samaneh Mahdavifar et al. 'Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning'. In: *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress*

(DASC/PiCom/CBDCom/CyberSciTech). Aug. 2020, pp. 515–522. DOI: 10.1109/DASC-PiCom-CBDCom-CyberSciTech49142.2020.00094.

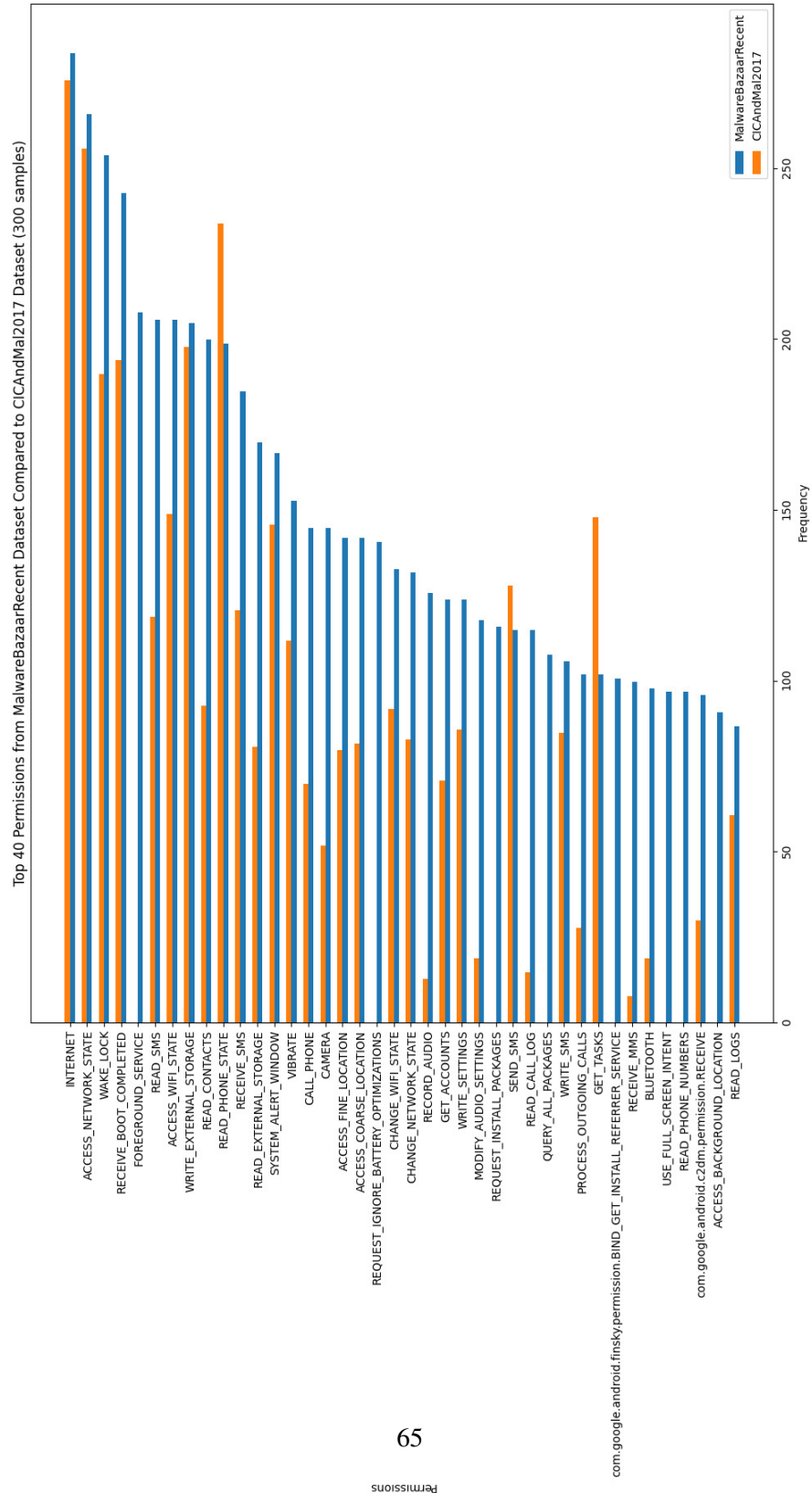
- [43] ElMouatez Billah Karbab et al. ‘MalDozer: Automatic framework for android malware detection using deep learning’. en. In: *Digital Investigation* 24 (Mar. 2018), S48–S59. ISSN: 17422876. DOI: 10.1016/j.diin.2018.01.007. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1742287618300392> (visited on 02/08/2023).
- [44] Sen Chen et al. *An Empirical Assessment of Security Risks of Global Android Banking Apps*. arXiv:1805.05236 [cs]. Feb. 2020. URL: <http://arxiv.org/abs/1805.05236> (visited on 06/09/2023).
- [45] Google. *Android Runtime (ART) and Dalvik*. en. 2022. URL: <https://source.android.com/docs/core/runtime> (visited on 18/06/2023).
- [46] Dora and Arthur. *Android Runtime Environment: DVM vs ART*. en. 2020. URL: <https://intexsoft.com/blog/android-runtime-environment-dvm-vs-art/> (visited on 18/06/2023).
- [47] *API Differences between 33 and 34*. URL: https://developer.android.com/sdk/api_diff/34/changes (visited on 02/08/2023).
- [48] Google. *<uses-sdk>*. en. URL: <https://developer.android.com/guide/topics/manifest/uses-sdk-element> (visited on 02/08/2023).
- [49] Google. *Manifest.permission*. en. URL: <https://developer.android.com/reference/android/Manifest.permission> (visited on 03/09/2023).
- [50] *Permissions on Android*. en. URL: <https://developer.android.com/guide/topics/permissions/overview> (visited on 16/08/2023).
- [51] *Dalvik bytecode*. en. URL: <https://source.android.com/docs/core/runtime/dalvik-bytecode> (visited on 04/09/2023).
- [52] *MalwareBazaar | Malware sample exchange*. URL: <https://bazaar.abuse.ch/> (visited on 28/08/2023).
- [53] VirusTotal. *Advanced corpus search*. en. URL: <https://developers.virustotal.com/reference/intelligence-search> (visited on 20/08/2023).
- [54] MalwareBazaar. *MalwareBazaar | FAQ*. URL: <https://bazaar.abuse.ch/faq/> (visited on 04/09/2023).
- [55] VirusShare. *VirusShare.com*. URL: https://virusshare.com/apiv2_reference (visited on 04/09/2023).
- [56] *Apktool*. en. URL: <https://apktool.org/> (visited on 20/08/2023).
- [57] Python. *os — Miscellaneous operating system interfaces*. URL: <https://docs.python.org/3/library/os.html> (visited on 11/09/2023).

- [58] *Androguard*. original-date: 2014-09-12T08:48:56Z. Aug. 2023. URL: <https://github.com/androguard/androguard> (visited on 20/08/2023).
- [59] Yuanchun Li et al. 'DroidBot: a lightweight UI-guided test input generator for Android'. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press, May 2017, pp. 23–26. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.8. URL: <https://dl.acm.org/doi/10.1109/ICSE-C.2017.8> (visited on 21/08/2023).
- [60] *Mobile Security Framework (MobSF)*. original-date: 2015-01-31T04:36:01Z. Aug. 2023. URL: <https://github.com/MobSF/Mobile-Security-Framework-MobSF> (visited on 20/08/2023).
- [61] Python. *argparse* — *Parser for command-line options, arguments and sub-commands*. URL: <https://docs.python.org/3/library/argparse.html> (visited on 11/09/2023).
- [62] Python. *pickle* — *Python object serialization*. URL: <https://docs.python.org/3/library/pickle.html> (visited on 04/09/2023).
- [63] Apktool. *[BUG] Invalid CEN header when unzipping apk · Issue #3198 · iBotPeaches/Apktool*. en. URL: <https://github.com/iBotPeaches/Apktool/issues/3198> (visited on 22/08/2023).
- [64] Apktool. *Exception in thread "main" brut.androlib.AndrolibException: brut.directory.DirectoryException: java.util.zip.ZipException: zip END header not found · Issue #2324 · iBotPeaches/Apktool*. en. URL: <https://github.com/iBotPeaches/Apktool/issues/2324> (visited on 22/08/2023).
- [65] gskinner. *RegExr: Learn, Build, & Test RegEx*. URL: <https://regexpr.com/> (visited on 28/08/2023).
- [66] Python. *collections* — *Container datatypes*. URL: <https://docs.python.org/3/library/collections.html> (visited on 04/09/2023).
- [67] Matplotlib. *Matplotlib* — *Visualization with Python*. URL: <https://matplotlib.org/> (visited on 04/09/2023).
- [68] Anthony Desnos, Geoffroy Gueguen and Sebastian Bachmann. *androguard package — Androguard 3.4.0 documentation*. URL: <https://androguard.readthedocs.io/en/latest/api/androguard.html> (visited on 03/09/2023).
- [69] *timeit* — *Measure execution time of small code snippets*. URL: <https://docs.python.org/3/library/timeit.html> (visited on 07/09/2023).
- [70] VirusShare. URL: https://virusshare.com/hashfiles/VirusShare_00476.md5 (visited on 11/09/2023).
- [71] Thomas Sutter. *Simple Spyware: Androids Invisible Foreground Services and How to (Ab)use Them*. arXiv:2011.14117 [cs]. Nov. 2020. URL: <http://arxiv.org/abs/2011.14117> (visited on 04/09/2023).

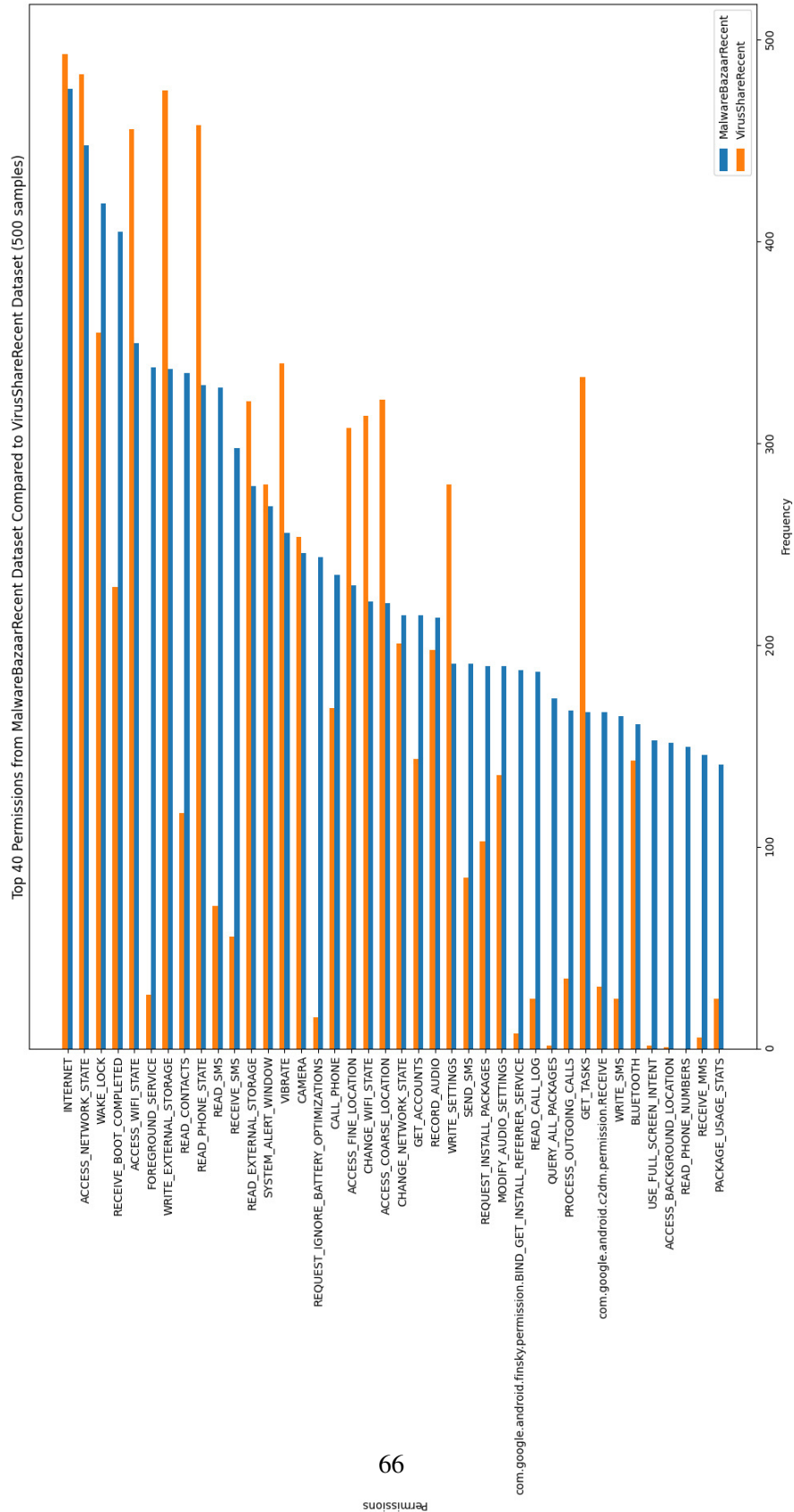
- [72] Google. *Optimize for Doze and App Standby | App quality*. en. URL: <https://developer.android.com/training/monitoring-device-state/doze-standby> (visited on 04/09/2023).
- [73] Haoyu Wang et al. 'RmvDroid: Towards A Reliable Android Malware Dataset with App Metadata'. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. ISSN: 2574-3864. May 2019, pp. 404–408. DOI: 10.1109/MSR.2019.00067.
- [74] Google. *AndroidX overview | Jetpack*. en. URL: <https://developer.android.com/jetpack/androidx> (visited on 04/09/2023).
- [75] Google. *Stable Releases | Jetpack*. en. URL: <https://developer.android.com/jetpack/androidx/versions/stable-channel> (visited on 04/09/2023).
- [76] The MITRE Corporation. *Event Triggered Execution: Broadcast Receivers, Sub-technique T1624.001 - Mobile | MITRE ATT&CK®*. URL: <https://attack.mitre.org/techniques/T1624/001/> (visited on 10/09/2023).
- [77] Google. *Uri*. en. URL: <https://developer.android.com/reference/android/net/Uri> (visited on 10/09/2023).
- [78] Google. *FileInputStream | Android Developers*. URL: <https://developer.android.com/reference/java/io/FileInputStream> (visited on 10/09/2023).
- [79] Google. *SharedPreferences*. en. URL: <https://developer.android.com/reference/android/content/SharedPreferences> (visited on 10/09/2023).
- [80] OkHttp. *Overview - OkHttp*. URL: <https://square.github.io/okhttp/> (visited on 11/09/2023).
- [81] The MITRE Corporation. *Exfiltration Over Alternative Protocol: Exfiltration Over Unencrypted Non-C2 Protocol, Sub-technique T1048.003 - Enterprise | MITRE ATT&CK®*. URL: <https://attack.mitre.org/techniques/T1048/003/> (visited on 11/09/2023).
- [82] Talos. *DoNot's Firestarter abuses Google Firebase Cloud Messaging to spread*. en. Oct. 2020. URL: <https://blog.talosintelligence.com/donot-firestarter/> (visited on 11/09/2023).
- [83] Malicia Lab. *AVClass*. original-date: 2016-07-01T16:57:31Z. Sept. 2023. URL: <https://github.com/malicialab/avclass> (visited on 10/09/2023).
- [84] *PEP 8 – Style Guide for Python Code | peps.python.org*. URL: <https://peps.python.org/pep-0008/> (visited on 11/09/2023).

A Appendix: Permissions Frequency Distribution Plots

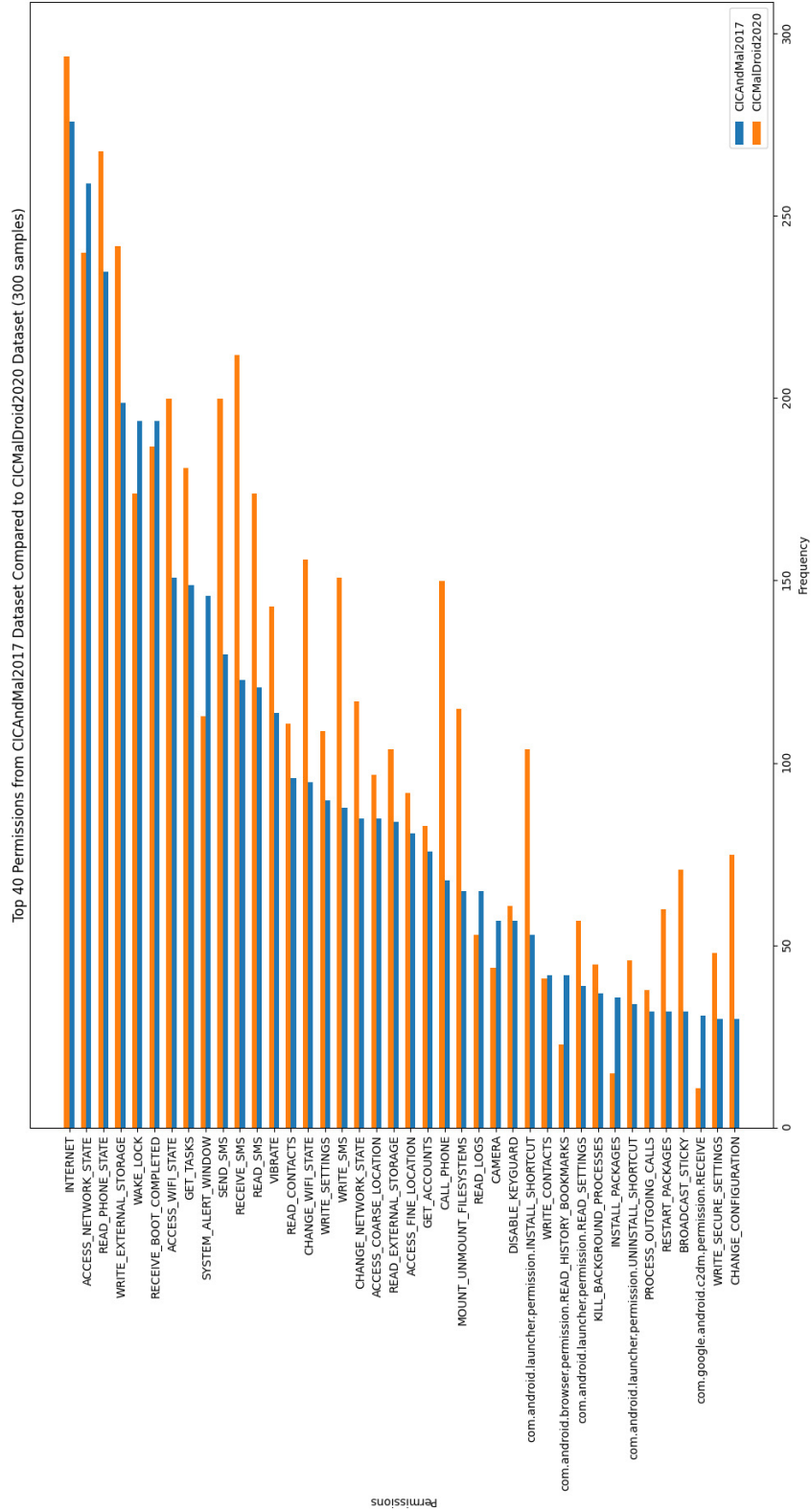
A.1 MalwareBazaarRecent to CICAndMal2017



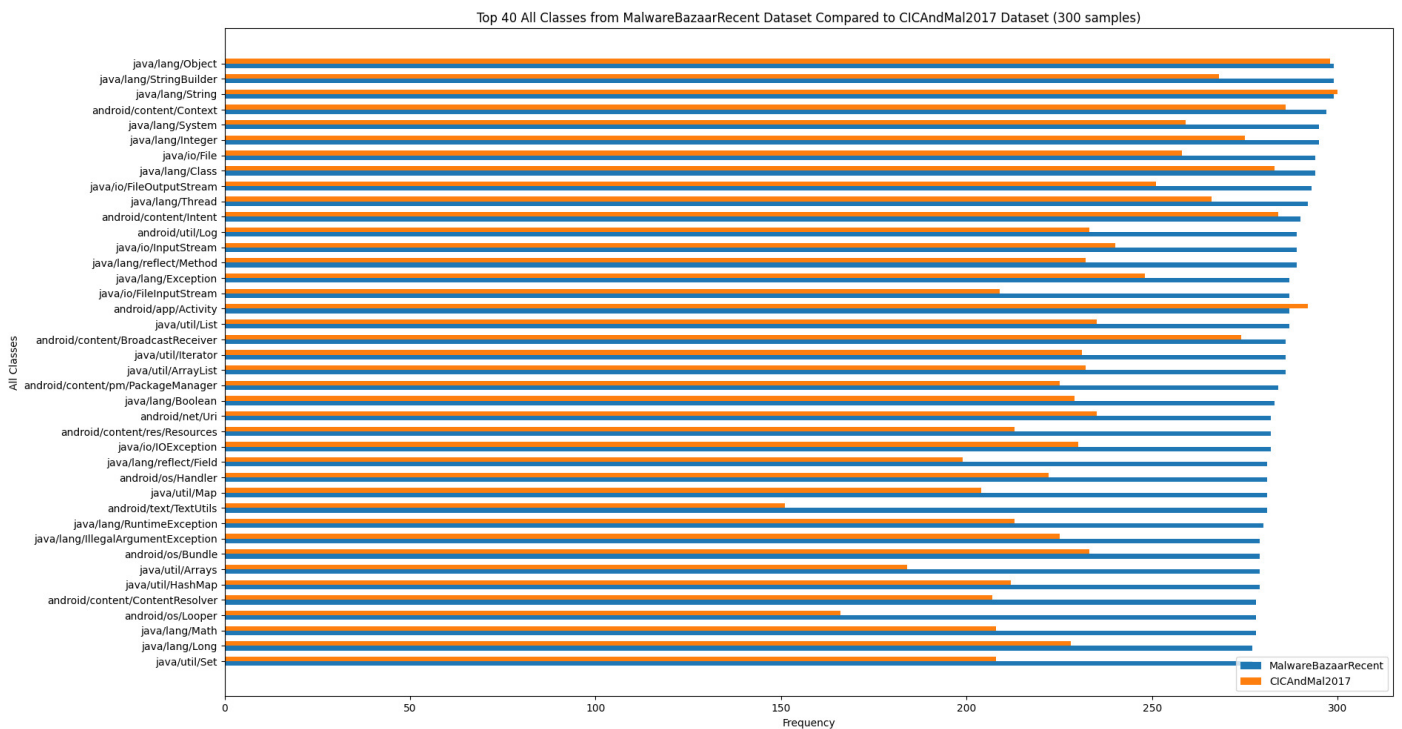
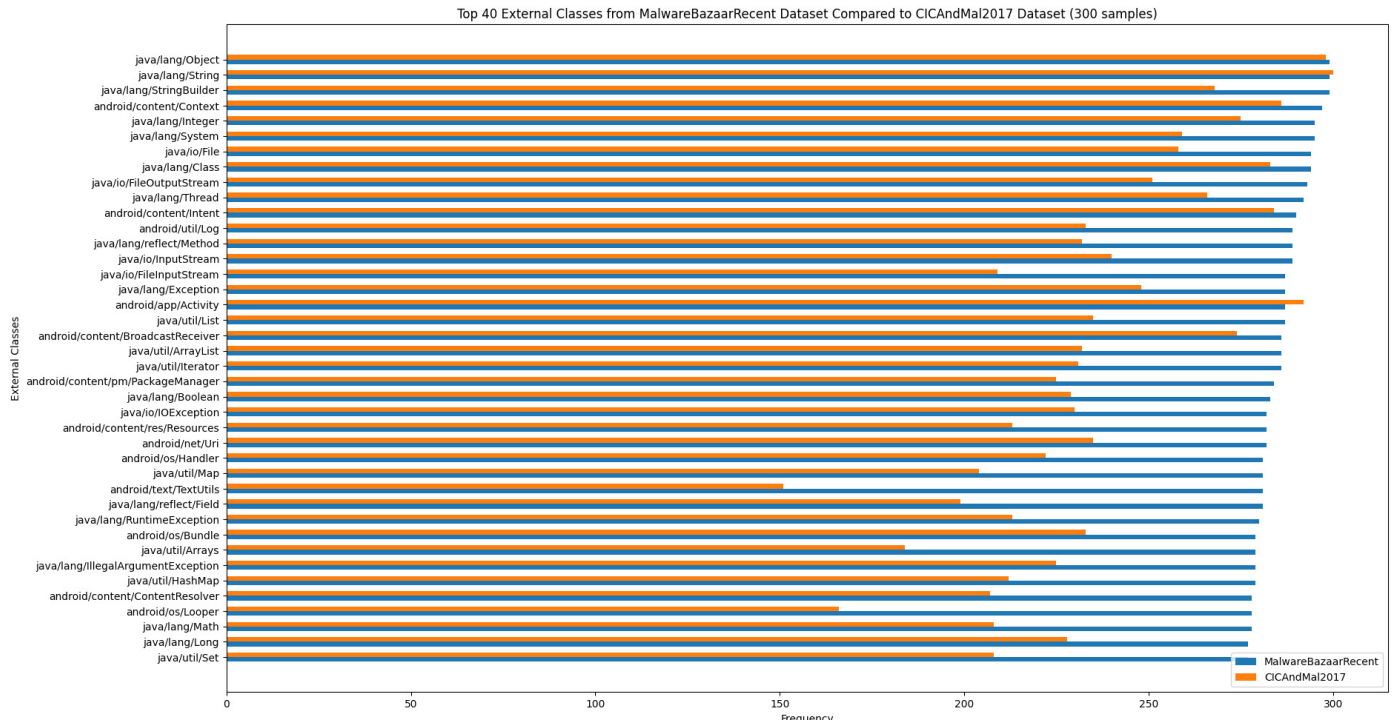
A.2 MalwareBazaarRecent to VirusShareRecent



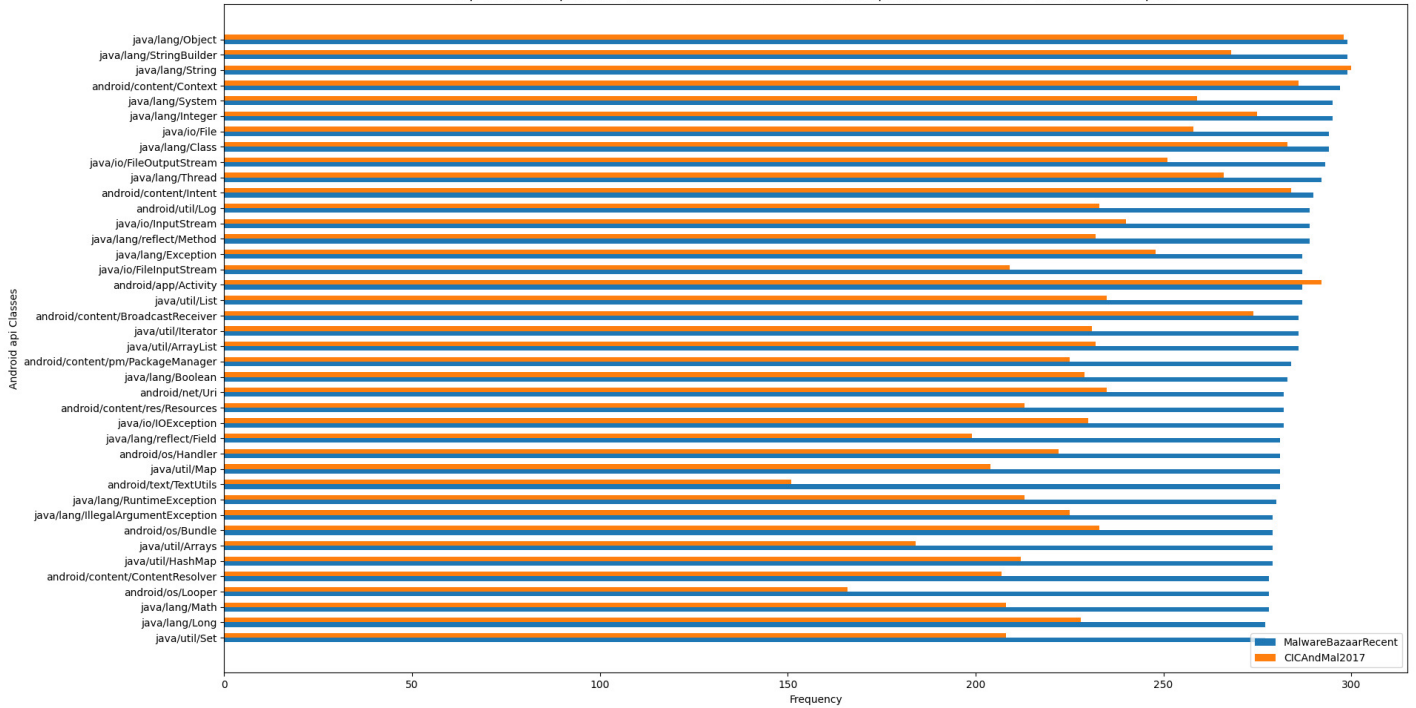
A.3 CICAndMal2017 to CICMalDroid2020



B Appendix: External, All and Android API Classes Plot Comparison

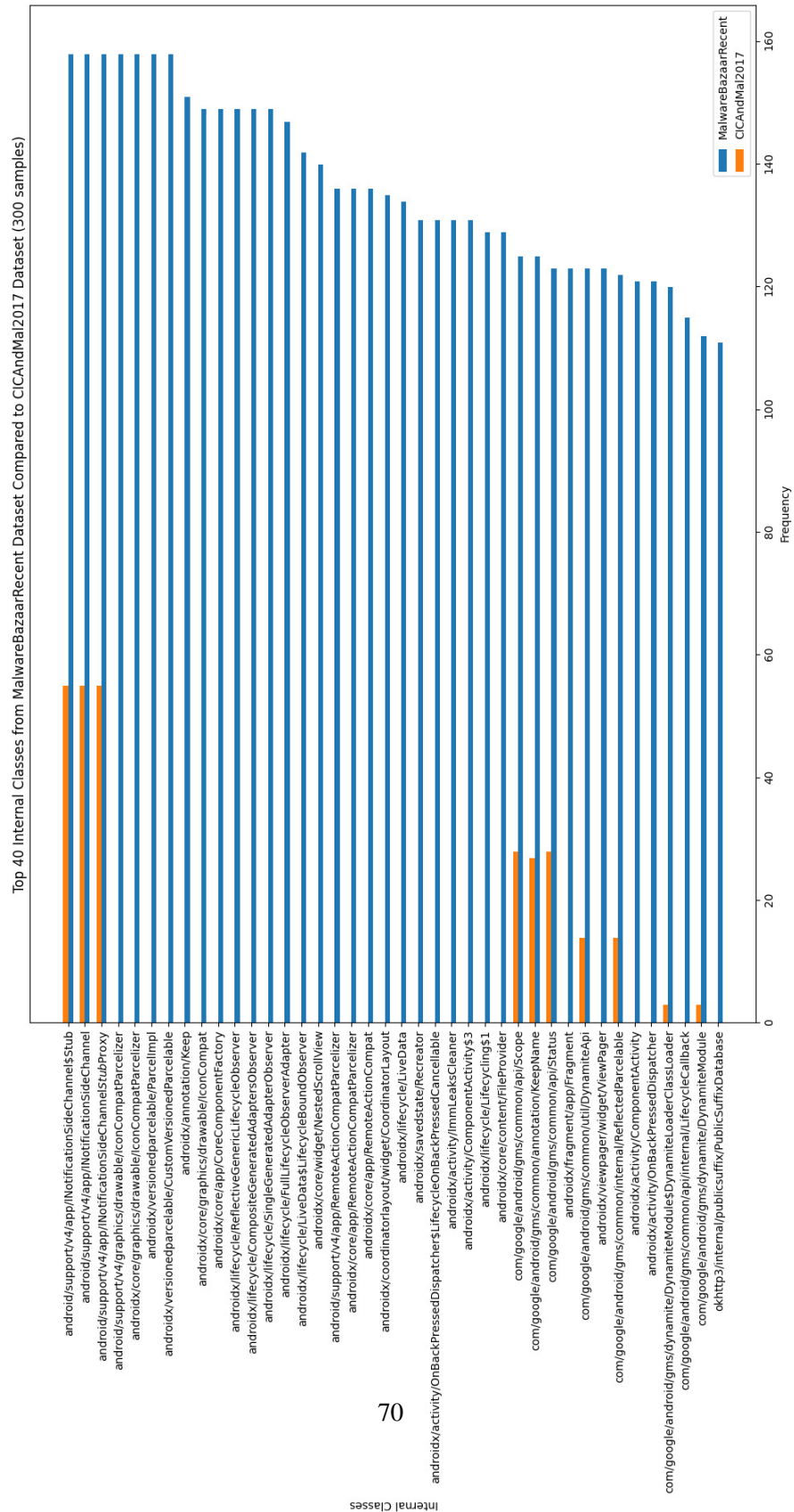


Top 40 Android api Classes from MalwareBazaarRecent Dataset Compared to CICAndMal2017 Dataset (300 samples)

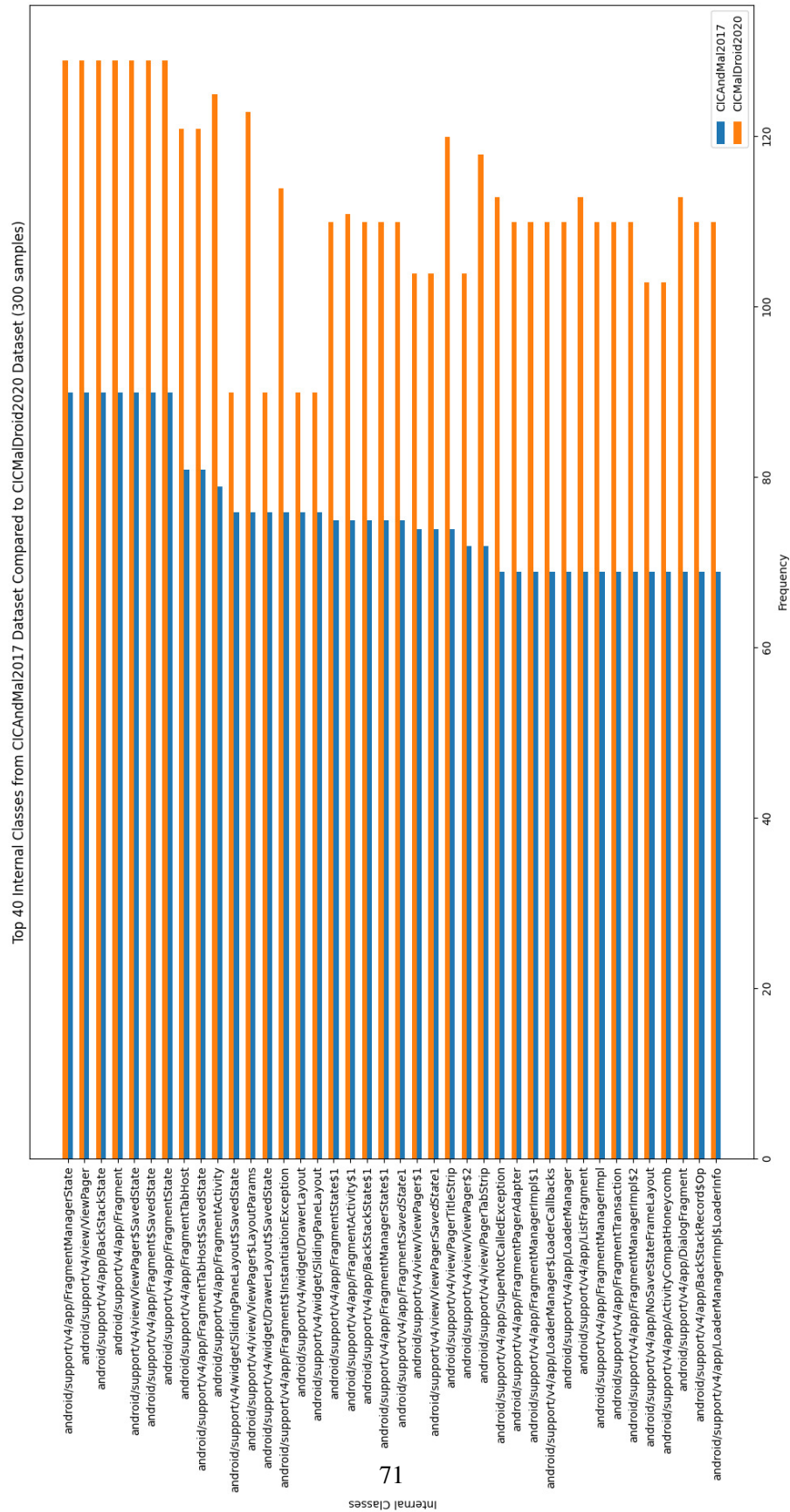


C Appendix: Internal Classes Frequency Distribution Plots

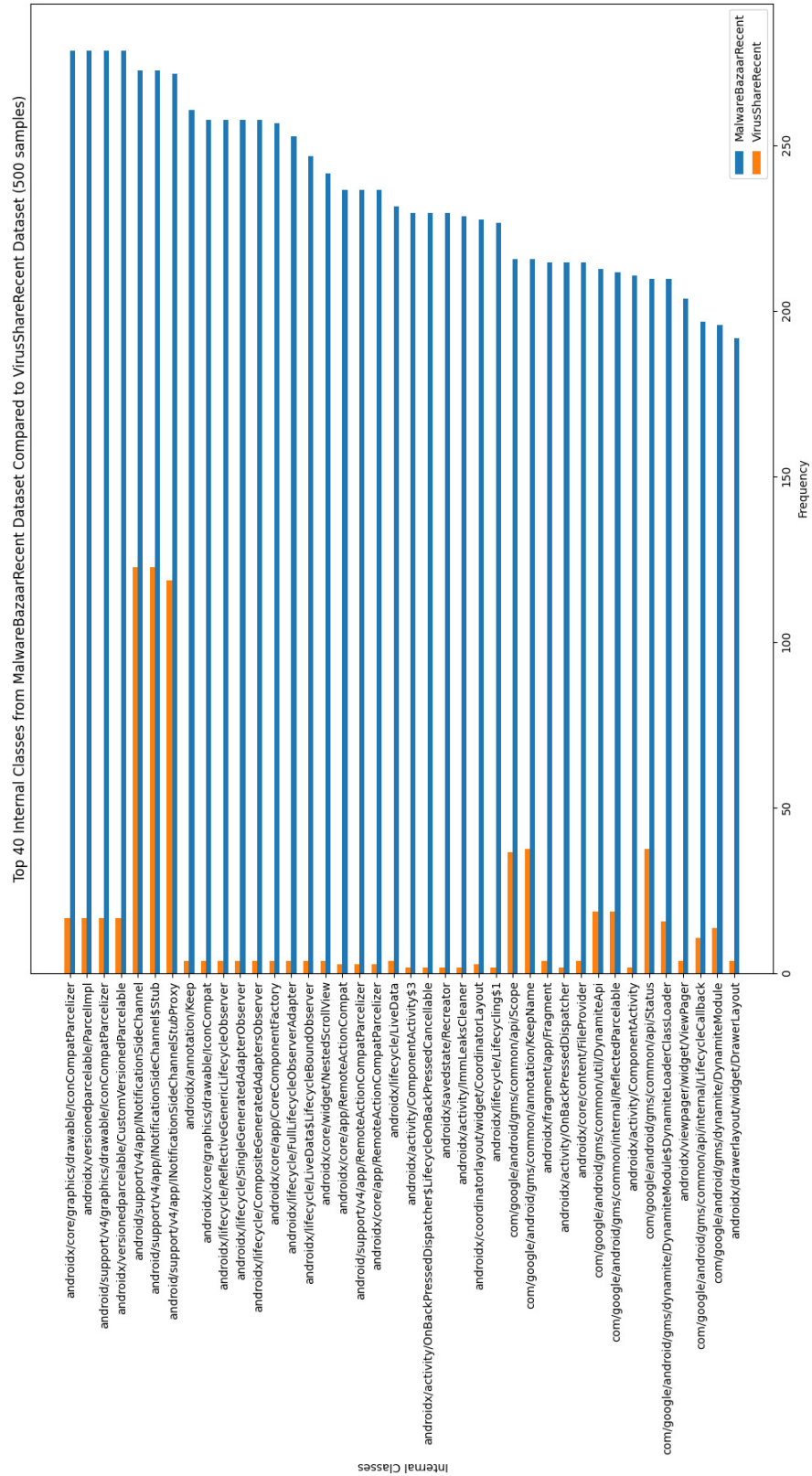
C.1 MalwareBazaarRecent to CICAndMal2017



C.2 MalwareBazaarRecent to VirusShareRecent

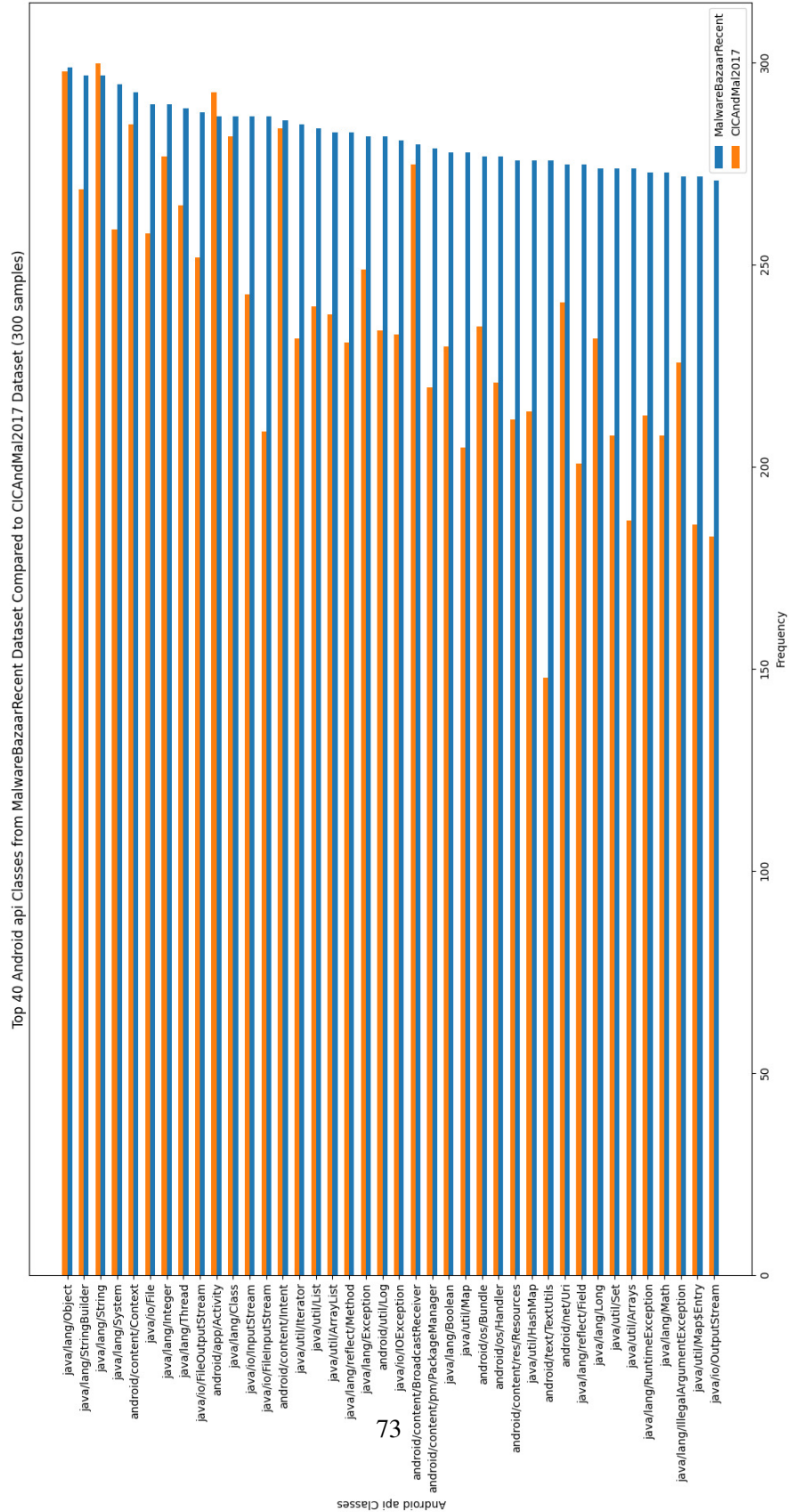


C.3 CICAndMal2017 to CICMalDroid2020

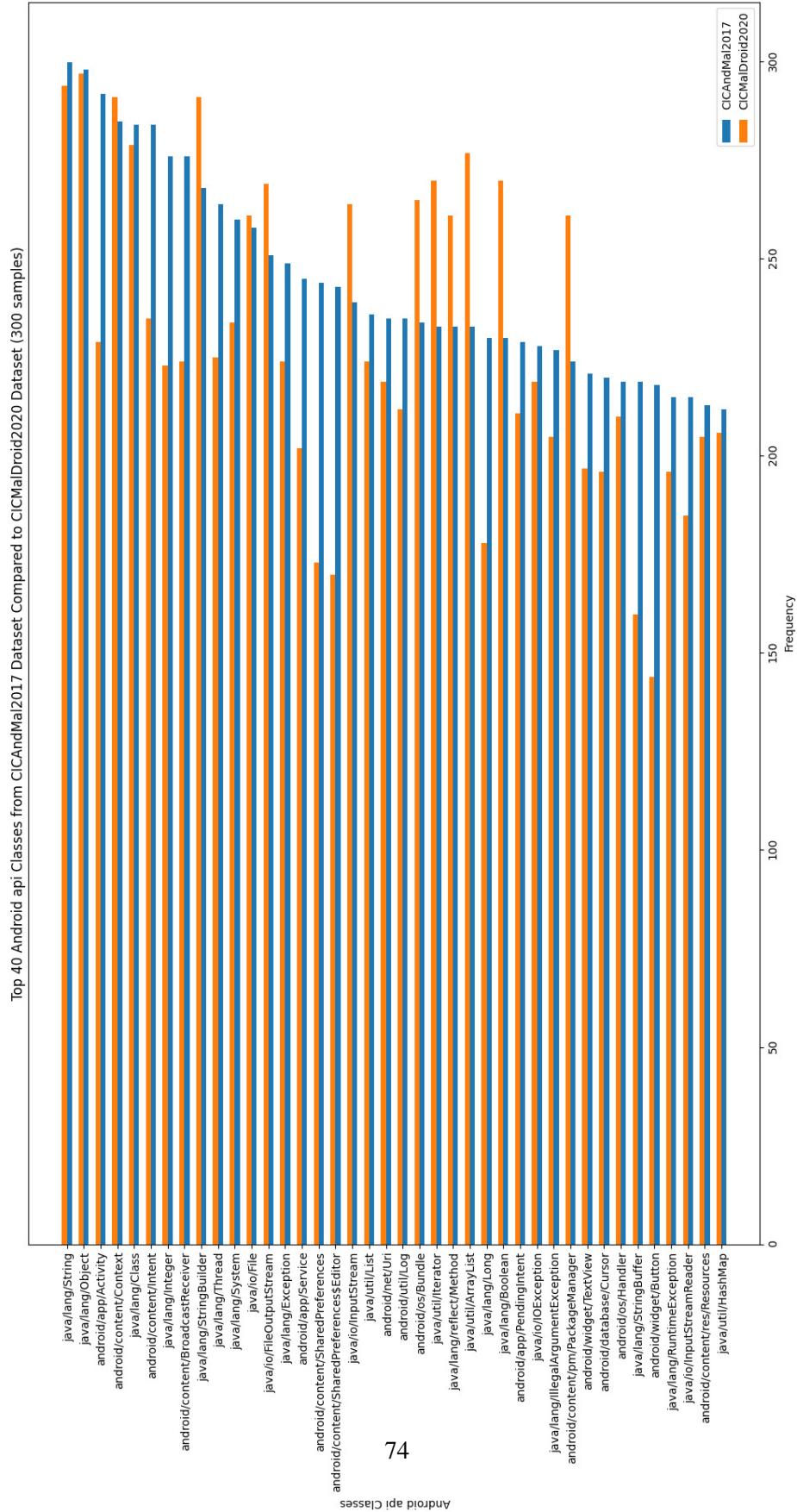


D Appendix: Android API Classes Frequency Distribution Plots

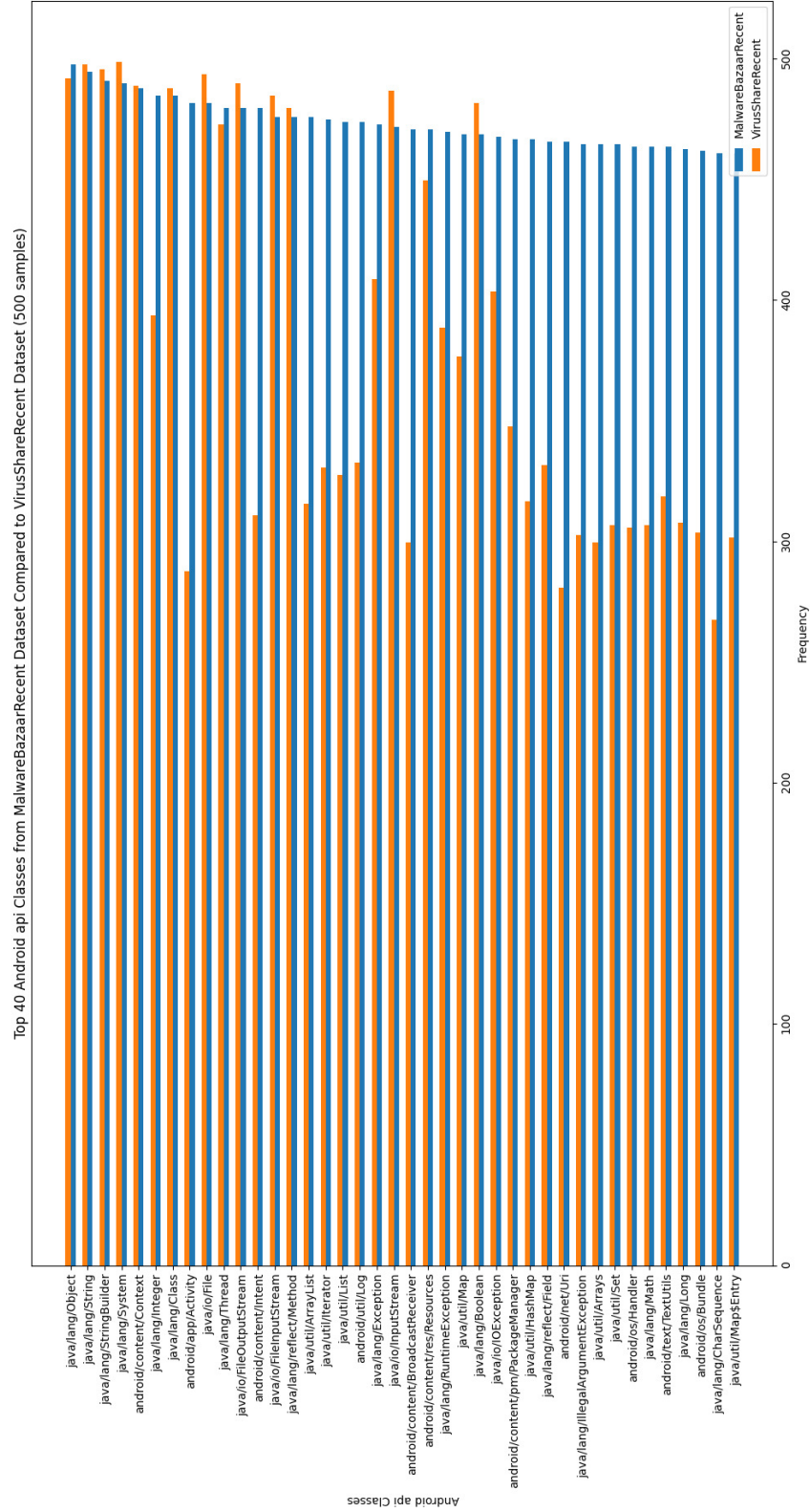
D.1 MalwareBazaarRecent to CICAndMal2017



D.2 MalwareBazaarRecent to VirusShareRecent

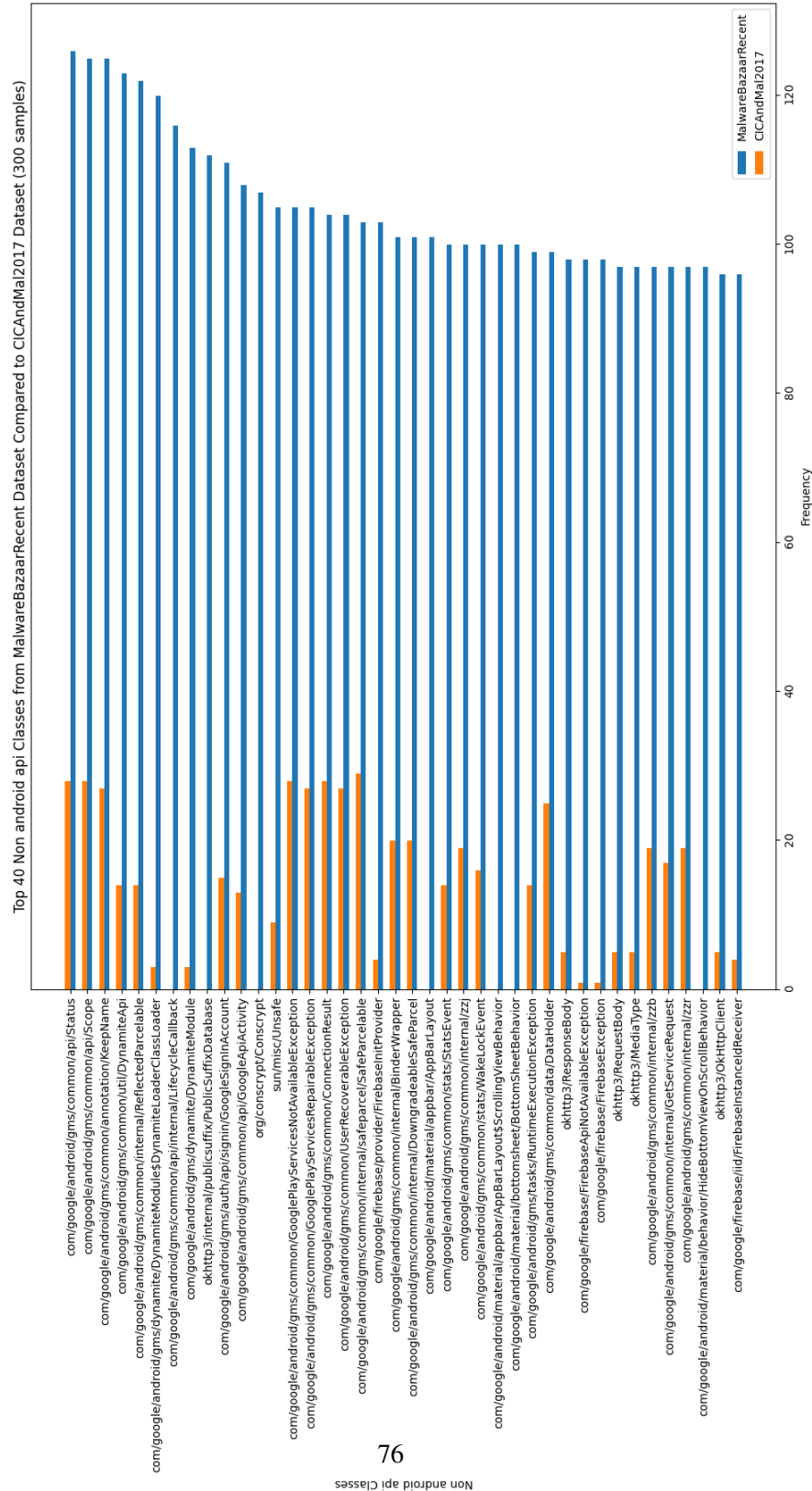


D.3 CICAndMal2017 to CICMalDroid2020

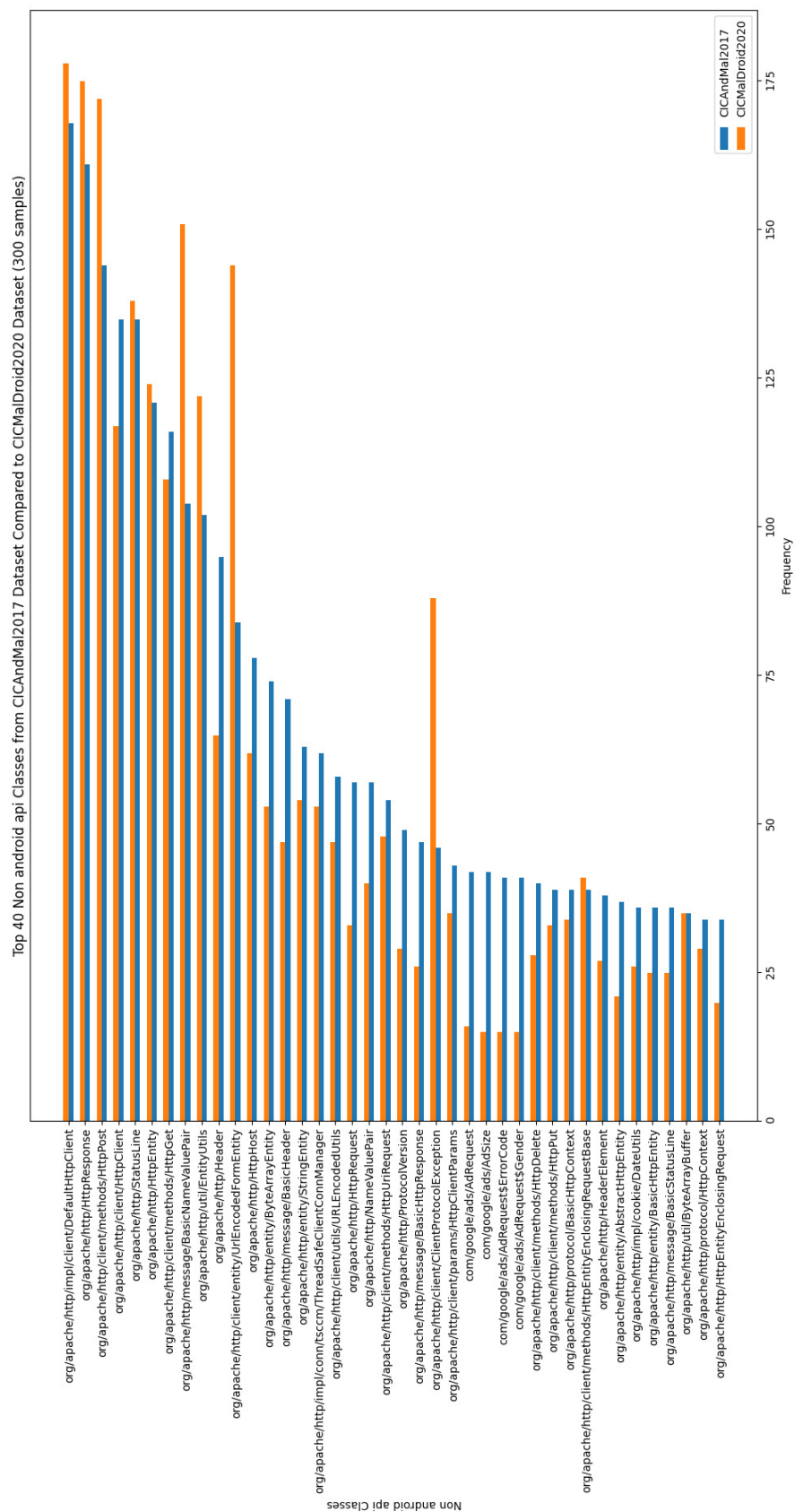


E Appendix: Non Android API Classes Frequency Distribution Plots

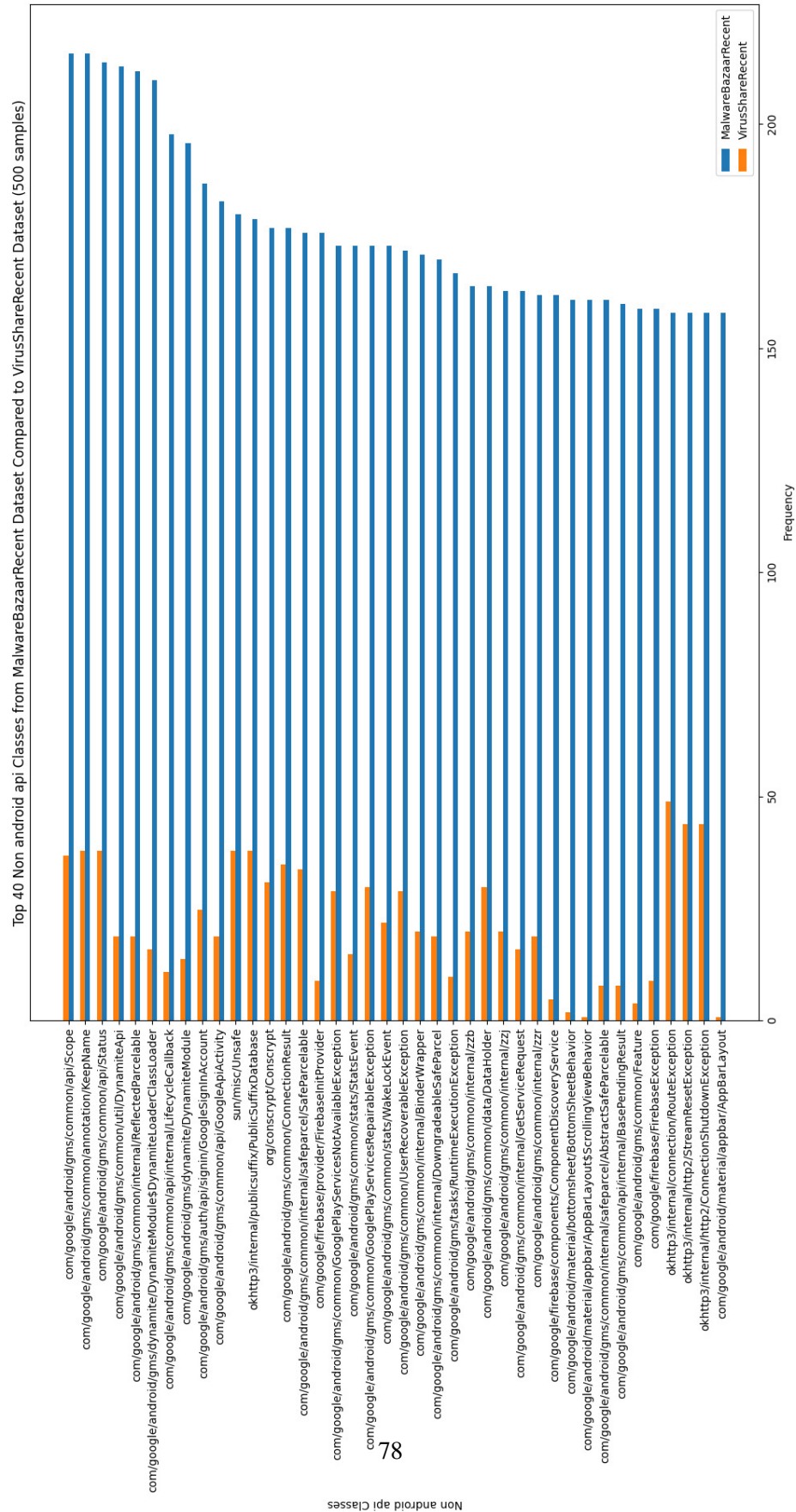
E.1 MalwareBazaarRecent to CICAndMal2017



E.2 MalwareBazaarRecent to VirusShareRecent



E.3 CICAndMal2017 to CICMalDroid2020



F Appendix: Droidloader Code

F.1 droidloader.py

```
1 import requests
2 import re
3 import json
4 import pyzipper
5 import argparse
6 import os
7 import sys
8 import time
9 import random
10 from configparser import ConfigParser
11
12 from core.classAnalyser import getClasses
13 from core.filterHashes import generateApkHashes
14 from core.dataProcess import readData, writeData
15 from graphAnalyser import processPermissions, processClasses
16
17 config = ConfigParser()
18 config.read('config.ini')
19 VTAPIKEY = config['VirusTotal']['apikey']
20 VSAPIKEY = config['VirusShare']['apikey']
21 MBURL = config['MalwareBazaar']['url']
22
23 def getHash(query):
24     response = requests.post(MBURL, data=query)
25     data = response.json()["data"]
26     hashes = []
27     for i in range(len(data)):
28         sample = data[i]
29         if sample["file_type"] == "apk":
30             hashes.append(sample["sha256_hash"])
31     return hashes
32
33 def downloadSample(dir, hash):
34     query = {"query": "get_file", "sha256_hash": hash}
35     response = requests.post(MBURL, data=query, timeout=60,
36                             allow_redirects=True)
37     open(f"{dir}/{hash}.zip", "wb").write(response.content)
38
39 def downloadSampleVS(dir, hash):
40     time.sleep(15)
41     url = f"https://virusshare.com/apiv2/download?apikey={VSAPIKEY}&hash={hash}"
42     response = requests.get(url)
43     open(f"{dir}/{hash}.zip", "wb").write(response.content)
```



```

44 def getMalwareByFamily(families, limit):
45     hashes = []
46     for family in families:
47         query = {"query": "get_siginfo", "signature": family, "limit":
limit}
48         hashes.extend(getHash(query))
49     return hashes
50
51 def getRecentMalware(limit):
52     query = {"query": "get_file_type", "file_type": "apk", "limit": limit
}
53     return getHash(query)
54
55 def extract(targetdir, outputdir):
56     os.mkdir(outputdir)
57     for archive in os.listdir(targetdir):
58         if archive.endswith(".zip"):
59             with pyzipper.AESZipFile(f"{targetdir}/{archive}") as zf:
60                 zf.extractall(path=outputdir, pwd=bytes("infected", "utf
-8"))
61     for file in os.listdir(outputdir):
62         if not file.endswith(".apk"):
63             os.rename(f"{outputdir}/{file}", f"{outputdir}/{file}")
64
65
66 def getReport(targetdir, outputfile):
67     url = "https://www.virustotal.com/api/v3/files/"
68     for _, _, files in os.walk(targetdir):
69         for sample in files:
70             if sample.endswith(".apk") or sample.endswith(".zip"):
71                 hash = sample[:-4]
72                 response = requests.get(f"{url}{hash}", headers={"accept"
: "application/json", "x-apikey": VTAPIKEY})
73                 with open(outputfile, "a") as f:
74                     json.dump(response.json(), f, ensure_ascii=False)
75                     f.write("\n")
76
77 def decompile(targetdir, outputdir, apktool):
78     samples = []
79     for root, _, files in os.walk(targetdir):
80         for file in files:
81             samples.append(os.path.abspath(os.path.join(root, file)))
82     os.mkdir(outputdir)
83     apktool = os.path.abspath(apktool)
84     for sample in samples:
85         if sample.endswith(".apk"):
86             os.system(f"cd {outputdir} && java -jar {apktool} d {sample}"
)
87

```

```

88 def cleanDecodedSamples(targetdir):
89     sampleCount = 0
90     fileCount = 0
91     for decodedSample in os.listdir(targetdir):
92         fileCount += 1
93         decodedSample = os.path.join(targetdir, decodedSample)
94         if not os.listdir(decodedSample):
95             os.rmdir(decodedSample)
96         else:
97             sampleCount += 1
98     print(f"[+] {fileCount} Files processed")
99     print(f"[+] {sampleCount} Files successfully decompiled")
100    print(f"[-] {fileCount - sampleCount} Invalid decompiled files
    removed")
101
102 def getPermissions(targetDir, limit):
103     permissionsList = []
104     dataset = os.listdir(targetDir)
105     random.shuffle(dataset)
106     for sourceCode in dataset[:limit]:
107         permissions = []
108         manifest = f"{targetDir}/{sourceCode}/AndroidManifest.xml"
109         if os.path.isfile(manifest):
110             with open(manifest) as f:
111                 permissions = re.findall(r'<uses-permission.*?>', f.
    read()) #Regex to extraction permissions from AnroidManifest
112                 permissions = [re.search(r'".*?"', permission).group()
    [1:-1] for permission in permissions if re.search(r'".*?"',
    permission)]
113                 permissionsList.append({"hash": sourceCode, "permissions": set(
    permissions)})
114     return permissionsList
115
116
117 if __name__ == "__main__":
118     parser = argparse.ArgumentParser(description="Dataset Generator",
    formatter_class=argparse.ArgumentDefaultsHelpFormatter)
119     subparsers = parser.add_subparsers(title="subcommand", dest="
    subcommand", description="subcommand")
120
121     downloadParser = subparsers.add_parser("download", help="Download
    Recent Malware samples from Virus Sample hosting platforms")
122     downloadParser.add_argument('outputfolder', type=str, help="Output
    Folder")
123     downloadParser.add_argument('-m', choices=['MalwareBazaarRecent', '
    MalwareBazaarFamily', 'VirusShareRecent'], help="Download Type", dest=
    "mode")
124     downloadParser.add_argument('-s', default=100, type=int, help="Number
    of Samples to Download", dest="size")

```

```

125
126     extractParser = subparsers.add_parser("extract", help="Extract folder
127         containing zipped samples")
128     extractParser.add_argument('targetfolder', type=str, help="Target
129         Folder")
130     extractParser.add_argument('-o', help="Output Folder", dest="
131         outputfolder")
132
133     decompileParser = subparsers.add_parser("decompile", help="Decompile
134         folder containg APK samples")
135     decompileParser.add_argument('targetfolder', type=str, help="Target
136         Folder")
137     decompileParser.add_argument('-o', help="Output Folder", dest="
138         outputfolder")
139
140     filterParser = subparsers.add_parser("filter", help="Filter a list of
141         MD5 from VirusShare for APK files")
142     filterParser.add_argument('targetfile', type=str, help="VirusShare
143         MD5 File")
144
145     permissionParser = subparsers.add_parser("permissions", help="Get
146         Permissions based off decompiled datasets")
147     permissionParser.add_argument('targetfolder', type=str, help="Target
148         Decoded Sample Folder")
149     permissionParser.add_argument('-s', default=10, type=int, help="
150         Number of Samples to get Permissions", dest="size")
151
152     classesParser = subparsers.add_parser("classes", help="Extracts
153         classes from decompiled datasets")
154     classesParser.add_argument('targetfolder', type=str, help="Target
155         Decoded Sample Folder")
156     classesParser.add_argument('-s', default=10, type=int, help="Number
157         of Samples to get Classes", dest="size")
158
159     graphParser = subparsers.add_parser("graph", help="Produce Graphs
160         from two datasets")
161     graphParser.add_argument('targetdata', type=str, help="Target Data")
162     graphParser.add_argument('-t', choices=['permissions', 'classes'],
163         help="Type of Feature", dest="type")
164     graphParser.add_argument('-d', help="Comparison Data", dest="
165         comparedata")
166     graphParser.add_argument('-s', default=100, type=int, help="Number of
167         Samples Retrieve in each Dataset", dest="size")
168     graphParser.add_argument('-c', default=20, type=int, help="Number of
169         Categories to Display", dest="categorySize")
170
171     if len(sys.argv)==1:
172         parser.print_help()
173         parser.exit()

```

```

155     args = parser.parse_args()
156
157     if args.subcommand == "download":
158         hashes = []
159         if args.mode == "MalwareBazaarFamily":
160             hashes = getMalwareByFamily(["Cerberus", "Hydra", "FluBot", "
161             Octo", "Ermac"], args.size)
162         if args.mode == "MalwareBazaarRecent":
163             hashes = getRecentMalware(args.size)
164         if args.mode == "VirusShareRecent":
165             data = readData("data/VirusShare/VirusShare_00476.pkl")
166             if data:
167                 hashes = data['hashes'][:args.size]
168         else:
169             print("No mode specified")
170         outputdir = args.outputfolder
171         if not os.path.exists(outputdir):
172             os.mkdir(outputdir)
173         if hashes:
174             for hash in hashes:
175                 if args.mode == "VirusShareRecent":
176                     downloadSampleVS(outputdir, hash)
177                 else:
178                     downloadSample(outputdir, hash)
179         else:
180             print("Directory Exists")
181
182     if args.subcommand == "extract":
183         outputdir = args.outputfolder
184         if args.outputfolder is not None:
185             if not os.path.exists(outputdir):
186                 extract(args.targetfolder, outputdir)
187             else:
188                 print("Directory Exists")
189         else:
190             print("Output Folder Required (-o)")
191
192     if args.subcommand == "decompile":
193         outputdir = args.outputfolder
194         if outputdir is not None:
195             if not os.path.exists(outputdir):
196                 decompile(args.targetfolder, outputdir, "dependencies/
197                 apktool_2.8.1.jar")
198                 cleanDecodedSamples(outputdir)
199             else:
200                 print("Directory Exists")
201         else:
202             print("Required Output Folder")

```

```

202
203     if args.subcommand == "filter":
204         md5File = args.targetfile
205         if os.path.exists(md5File):
206             outputFileName = os.path.basename(md5File)[-4]
207             fileHashes = readData(f"data/VirusShare/{outputFileName}.pkl"
) # Get saved hashes
208             if fileHashes:
209                 apkHashes, count = generateApkHashes(md5File, fileHashes[
'hashes'], fileHashes['count'])
210             else:
211                 apkHashes, count = generateApkHashes(md5File, [], 0)
212             if apkHashes and count:
213                 writeData({"hashes": apkHashes, "count": count}, "
VirusShare", outputFileName)
214
215     if args.subcommand == "permissions":
216         targetdir = args.targetfolder
217         dirName = os.path.basename(targetdir)
218         if dirName == "":
219             dirName = os.path.basename(os.path.dirname(targetdir))
220         permissions = getPermissions(targetdir, args.size)
221         if permissions:
222             writeData(permissions, dirName, "permissions")
223         else:
224             print("Invalid Sample")
225
226     if args.subcommand == "classes":
227         targetdir = args.targetfolder
228         dirName = os.path.basename(targetdir)
229         if dirName == "":
230             dirName = os.path.basename(os.path.dirname(targetdir))
231         classes = getClasses(targetdir, args.size)
232         if classes:
233             writeData(classes, dirName, "classes")
234         else:
235             print("Invalid Sample")
236
237     if args.subcommand == "graph":
238         targetdata = args.targetdata
239         comparedata = args.comparedata
240         sampleSize = args.size
241         categorySize = args.categorySize
242         if args.type == "permissions":
243             processPermissions(f"{targetdata}/permissions.pkl", f"{
comparedata}/permissions.pkl", categorySize, sampleSize)
244         elif args.type == "classes":
245             processClasses(f"{targetdata}/classes.pkl", f"{comparedata}/
classes.pkl", categorySize, sampleSize)

```

```

246         else:
247             print("No type specified")

```

F.2 core/classAnalyser.py

```

1  import os
2  import re
3  import random
4
5  def formatClass(item):
6      if "const-class" in item:
7          return item.split(" ")[2][:-1]
8      if "}," in item and ";->" in item:
9          return item.split("},")[1].split(";")[0].replace(" ", "")
10     else:
11         return ""
12
13 def className(root, smaliFile):
14     smaliCode = os.path.join(root, smaliFile)
15     with open(smaliCode, "r") as f:
16         name = re.findall((r"\.class.*"), f.read())[0]
17         name = name.split()
18         name = name[len(name)-1][1:-1]
19     return name
20
21 def returnClasses(root, smaliFile):
22     smaliCode = os.path.join(root, smaliFile)
23     with open(smaliCode, "r") as f:
24         classes = re.findall(r"invoke-virtual.*|invoke-super.*|invoke-
direct.*|invoke-static.*|invoke-interface.*|const-class.*", f.read())
25         if classes:
26             classes = [formatClass(item)[1:] if formatClass(item).
startswith("L") else formatClass(item)[2:] for item in classes if item
!= ""]
27     return classes
28
29 def getSmaliFolders(sourceCode):
30     return [file for file in sourceCode if re.search(r'smali_classes\d+|
smali', file)]
31
32 def getClasses(targetDir, limit):
33     classesList = []
34     decodedSamples = os.listdir(targetDir)
35     random.shuffle(decodedSamples)
36     i = 0
37     for application in decodedSamples[:limit]:
38         isApplication = False
39         classes = set()
40         internalClasses = set()
41         sourceCodePath = os.path.join(targetDir, application)

```

```

42     sourceCode = os.listdir(sourceCodePath)
43     smaliFolders = getSmaliFolders(sourceCode)
44     i += 1
45     for folders in smaliFolders:
46         for root, _, files in os.walk(os.path.join(sourceCodePath,
47             folders)):
48             isApplication = True
49             for smaliFile in files:
50                 if(smaliFile.endswith(".smali")):
51                     internalClasses.add(className(root, smaliFile))
52                     classes.update(returnClasses(root, smaliFile))
53                     classes.discard("")
54             if isApplication:
55                 classInfo = {
56                     "hash": application,
57                     "internal": internalClasses,
58                     "external": classes - internalClasses,
59                     "all": classes | internalClasses
60                 }
61                 classesList.append(classInfo)
62     return classesList

```

F.3 core/filterHashes.py

```

1  import requests
2  import os
3  from concurrent.futures import ThreadPoolExecutor
4  from configparser import ConfigParser
5
6  config = ConfigParser()
7  config.read(os.path.abspath('config.ini'))
8  VTAPIKEY = config['VirusTotal']['apikey']
9
10 def getType(hash):
11     url = "https://www.virustotal.com/api/v3/files/"
12     response = requests.get(f"{url}{hash}", headers={"accept": "
13     application/json", "x-apikey": VTAPIKEY})
14     try:
15         sampleType = response.json()['data']['attributes']['
16         type_description']
17         extension = response.json()['data']['attributes']['type_extension
18         ',]
19         detections = response.json()['data']['attributes']['
20         last_analysis_stats']['malicious']
21         if sampleType == "Android" and extension == "apk" and detections
22         >= 2:
23             return hash
24         else:
25             return None
26     except:

```

```

22         return None
23
24 def listmd5(file):
25     with open(f"{file}") as f:
26         md5List = [md5.strip("\n") for md5 in f.readlines()[6:]]
27     return md5List
28
29 def generateApkHashes(md5File, apkHashes, count, limit=18000):
30     count = 0
31     md5List = listmd5(md5File)
32     executor = ThreadPoolExecutor(max_workers=4)
33     i = 0
34     if count < len(md5List):
35         for hash in executor.map(getType, md5List[count:count+limit]):
36             i+=1
37             if hash is not None:
38                 apkHashes.append(hash)
39         return apkHashes, count+limit
40     else:
41         print("MD5 File exhausted")
42         return False, False

```

F.4 core/dataProcess.py

```

1 import pickle
2 import os
3 import random
4 from collections import Counter
5
6 def writeData(data, dirName, fileName):
7     if not os.path.exists(f"data/{dirName}"):
8         os.makedirs(f"data/{dirName}")
9     with open(f"data/{os.path.basename(dirName)}/{fileName}.pkl", "wb")
10    as f:
11        pickle.dump(data, f)
12
13 def readData(fileName):
14     if os.path.exists(fileName):
15         with open(f"{fileName}", "rb") as f:
16             return pickle.load(f)
17     else:
18         return False
19
20 def average(numbers):
21     if len(numbers) == 0:
22         return 0
23     total = sum(numbers)
24     average = total / len(numbers)
25     return average

```



```

26 def getSampleSize(fileName):
27     data = readData(fileName)
28     if data:
29         return len(data)
30     else:
31         return False
32
33 def validateAPI(classes, AndroidAPI):
34     for package in AndroidAPI:
35         if classes.startswith(package):
36             return True
37     return False
38
39 def getPermissionsFrequency(fileName, size):
40     permissionFrequency = Counter()
41     data = readData(fileName)
42     if data:
43         random.shuffle(data)
44         for application in data[:size]:
45             permissionFrequency += Counter(application['permissions'])
46     return permissionFrequency
47
48 def getClassesFrequency(fileName, size):
49     classTypes = ['internal', 'external', 'all']
50     classFrequency = {'internal': Counter(), 'external': Counter(), 'all': Counter()}
51     data = readData(fileName)
52     if data:
53         random.shuffle(data)
54         for application in data[:size]:
55             for types in classTypes:
56                 classFrequency[types] += Counter(application[types])
57     return classFrequency

```

F.5 graphAnalyser.py

```

1 import matplotlib.pyplot as plt
2 from collections import Counter
3 from core.dataProcess import readData, getClassesFrequency,
4   getPermissionsFrequency, getSampleSize, validateAPI
5 import os
6 import numpy as np
7
8 def formatData(data, categories):
9     data = data.most_common(categories)
10    return [item[0] for item in data], [item[1] for item in data]
11
12 def processSample(file1, file2, size=None):
13     dataset = os.path.basename(os.path.dirname(file1))
14     dataset2 = os.path.basename(os.path.dirname(file2))

```

```

14     sampleSize = getSampleSize(file1)
15     sampleSize2 = getSampleSize(file2)
16     if size is None or size > sampleSize or size > sampleSize2:
17         size = sampleSize if sampleSize < sampleSize2 else sampleSize2
18     return dataset, dataset2, size
19
20
21 def processGraphData(category, featureType, categorySize, sampleSize,
22 frequency, frequency2, dataset, dataset2):
23     category = category[:-1]
24     frequency = frequency[:-1]
25     frequency2 = frequency2[:-1]
26     y = np.arange(len(category))
27     barWidth = 0.3
28     dataset = dataset.replace("Decoded", "")
29     dataset2 = dataset2.replace("Decoded", "")
30     plt.barh(y - barWidth/2, frequency, barWidth, label=f"{dataset}")
31     plt.barh(y + barWidth/2, frequency2, barWidth, label=f"{dataset2}")
32
33     plt.xlabel('Frequency')
34     plt.ylabel(f'{featureType}')
35     plt.title(f'Top {categorySize} {featureType} from {dataset} Dataset
36 Compared to {dataset2} Dataset ({sampleSize} samples)')
37     plt.yticks(y, category)
38     plt.legend()
39
40     plt.show()
41
42 def processPermissions(file1, file2, categorySize, sampleSize=None):
43     dataset, dataset2, size = processSample(file1, file2, sampleSize)
44     Data = getPermissionsFrequency(file1, size)
45     Data2 = getPermissionsFrequency(file2, size)
46     category, frequency = formatData(Data, categorySize)
47     frequency2 = []
48     for permission in category:
49         frequency2.append(Data2[permission])
50     print(frequency2)
51     category = [permission.split(".")[0] if ".".join(permission.split(".")
52 [0:2]) == "android.permission" else permission for permission in
53 category ]
54     processGraphData(category, "Permissions", categorySize, size,
55 frequency, frequency2, dataset, dataset2)
56
57 def processClasses(file1, file2, categorySize, sampleSize=None):
58     dataset, dataset2, size = processSample(file1, file2, sampleSize)
59     classTypes = ['internal', 'external', 'all', "Android API", "Non
60 Android API"]
61     androidAPI = readData('data/AndroidAPI.pkl')
62     categoryList = []

```

```

57     frequencyList = []
58     frequencyList2 = []
59     Data = getClassesFrequency(file1, size)
60     Data2 = getClassesFrequency(file2, size)
61     for types in classTypes:
62         frequency2 = []
63         if types == "Android API" or types == "Non Android API":
64             if types == "Android API":
65                 types = 'all'
66                 filteredData = {key: count for key, count in Data[types].
items() if validateAPI(key, androidAPI)}
67             else:
68                 types = 'all'
69                 filteredData = {key: count for key, count in Data[types].
items() if not validateAPI(key, androidAPI)}
70                 category, frequency = formatData(Counter(filteredData),
categorySize)
71             else:
72                 category, frequency = formatData(Data[types], categorySize)
73                 categoryList.append(category)
74                 frequencyList.append(frequency)
75                 for classes in category:
76                     frequency2.append(Data2[types][classes])
77                 frequencyList2.append(frequency2)
78     for i in range(5):
79         category = categoryList[i]
80         frequency = frequencyList[i]
81         frequency2 = frequencyList2[i]
82         processGraphData(category, f"{classTypes[i].capitalize()}
Classes", categorySize, size, frequency, frequency2, dataset, dataset2
)

```

G Appendix: Testing and Benchmarking Code

G.1 test_droidloader.py

```
1 from droidloader import getPermissions
2 from core.classAnalyser import getClasses
3 from core.androguardAnalyser import getPermissionsAndClasses
4
5 def testGetPermissions():
6     permissions = getPermissions('datasets/TestDecoded', 10)
7     androguardPermissions = getPermissionsAndClasses('datasets/Test', 10)
8     [0]
9     successful = 0
10    for i in range(len(permissions)):
11        for j in range(len(androguardPermissions)):
12            if permissions[i]['hash'] == androguardPermissions[j]['hash']:
13                successful += 1
14    print(f"[+] 10 samples tested")
15    print(f"[+] {successful} permissions from samples are correct
16    compared to Androguard")
17
18 def testGetClasses():
19     classes = getClasses('datasets/TestDecoded', 10)
20     androguardClasses = getPermissionsAndClasses('datasets/Test', 10)[1]
21     classTypes = ['internal', 'external', 'all']
22     equalScore = [0, 0, 0]
23     containsScore = [0, 0, 0]
24     difference = []
25     total = []
26     for i in range(len(classes)):
27         for j in range(len(androguardClasses)):
28             if classes[i]['hash'] == androguardClasses[j]['hash']:
29                 for k in range(len(classTypes)):
30                     if classes[i][classTypes[k]] == androguardClasses[j][
31                     classTypes[k]]:
32                         equalScore[k] += 1
33                     if len(androguardClasses[j][classTypes[k]] - classes[
34                     i][classTypes[k]]) == 0:
35                         containsScore[k] += 1
36
37     print(f"[+] 10 samples tested")
38     for i in range(len(classTypes)):
39         print(f"[+] {equalScore[i]} {classTypes[i]} classes from samples
40         are correct compared to Androguard")
41         print(f"[+] {containsScore[i]} {classTypes[i]} classes from
42         samples contains classes from Androguard")
43
44     print(total)
```

```

39     for i in range(len(total)):
40         print(f" Hash: {classes[i]['hash']} Total classes: {total[i]}
          Difference: {difference[i]}")
41
42 if __name__ == "__main__":
43     #testGetPermissions()
44     testGetClasses()

```

G.2 benchmarker.py

```

1  from core.androguardAnalyser import getPermissionsAndClasses
2  from core.classAnalyser import getClasses
3  from core.dataProcess import readData, writeData
4  from droidloader import getPermissions, decompile, cleanDecodedSamples
5  import timeit
6  import random
7  import os
8  import shutil
9
10 def generateSample(targetDir, size):
11     datasets=['datasets/MalwareBazaarRecent', 'datasets/CICAndMal2017', '
          datasets/VirusShareRecent', 'datasets/CICMalDroid2020']
12     samples = []
13     for dataset in datasets:
14         for root, _, files in os.walk(dataset):
15             for sample in files:
16                 samples.append(os.path.join(root, sample))
17     random.shuffle(samples)
18     samples = samples[:size]
19     for sample in samples:
20         shutil.copy(sample, targetDir)
21
22 def decompileTime(sampleDir, targetDir):
23     if os.path.exists(targetDir):
24         shutil.rmtree(targetDir)
25     decompile(sampleDir, targetDir, 'dependencies/apktool_2.8.1.jar')
26     cleanDecodedSamples(targetDir)
27
28 def permissionsAndClassesTime(sampleDir, size):
29     permissions = getPermissions(sampleDir, size)
30     classes = getClasses(sampleDir, size)
31     return permissions, classes
32
33 def permissionsAndClassesAndroguardTime(sampleDir, size):
34     permissions, classes = getPermissionsAndClasses(sampleDir, size)
35     return permissions, classes
36
37 if __name__ == "__main__":
38     sampleSize = [10, 25, 50]
39     times = readData('data/Benchmark/times.pkl')

```

```

40     if not times:
41         times = []
42     sampleDir = 'datasets/Benchmark'
43     decodedDir = 'datasets/BenchmarkDecoded'
44     for size in sampleSize:
45         if os.path.exists(sampleDir):
46             shutil.rmtree(sampleDir)
47         os.mkdir(sampleDir)
48         generateSample(sampleDir, size)
49         execTimeDecompile = timeit.timeit(lambda: decompileTime(sampleDir
, decodedDir), number=1)
50         execTime = timeit.timeit(lambda: permissionsAndClassesTime(
decodedDir, size), number=1)
51         execTimeAndroguard = timeit.timeit(lambda:
permissionsAndClassesAndroguardTime(sampleDir, size), number=1)
52         times.append({"sampleSize": size, "decompileTime":
execTimeDecompile, "droidloader": execTime, "androguard":
execTimeAndroguard})
53         writeData(times, "Benchmark", "times")

```

G.3 core/androguardAnalyser.py

```

1  from androguard.misc import AnalyzeAPK
2  import os
3
4  def analyseAPK(file):
5      apkFile, _, dx = AnalyzeAPK(file)
6      permissions = [permission for permission in apkFile.get_permissions()]
7      classes = [str(dexClass.name[1:-1]) for dexClass in dx.get_classes()]
8      externalClasses = [str(dexClass.name[1:-1]) for dexClass in dx.
get_external_classes()]
9      return permissions, classes, externalClasses
10
11 def getPermissionsAndClasses(targetDir, limit):
12     classesList = []
13     permissionsList = []
14     decodedSamples = os.listdir(targetDir)
15     for application in decodedSamples[:limit]:
16         permissions = set()
17         classes = set()
18         externalClasses = set()
19         permissions, apkClass, apkExternalClass = analyseAPK(os.path.join
(targetDir, application,))
20         classes.update(apkClass)
21         externalClasses.update(apkExternalClass)
22         classInfo = {
23             "hash": application[:-4],
24             "internal": classes - externalClasses,

```

```
25         "external": externalClasses,
26         "all": classes
27     }
28     permissionsList.append({"hash": application[:-4], "permissions":
set(permissions)})
29     classesList.append(classInfo)
30     return permissionsList, classesList
```

H Appendix: Droidloader Commands Options

```
graph                                     Produce graphs from the datasets
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py download
usage: droidloader.py download [-h] [-m {MalwareBazaarRecent,MalwareBazaarFamily,VirusShareRecent}] [-s SIZE] outputfolder
droidloader.py download: error: the following arguments are required: outputfolder
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py extract
usage: droidloader.py extract [-h] [-o OUTPUTFOLDER] targetfolder
droidloader.py extract: error: the following arguments are required: targetfolder
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py decompile
usage: droidloader.py decompile [-h] [-o OUTPUTFOLDER] targetfolder
droidloader.py decompile: error: the following arguments are required: targetfolder
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py filter
usage: droidloader.py filter [-h] targetfile
droidloader.py filter: error: the following arguments are required: targetfile
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py permissions
usage: droidloader.py permissions [-h] [-s SIZE] targetfolder
droidloader.py permissions: error: the following arguments are required: targetfolder
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py classes
usage: droidloader.py classes [-h] [-s SIZE] targetfolder
droidloader.py classes: error: the following arguments are required: targetfolder
alex@ubuntu:~/Documents/University-Dissertation$
```

```
alex@ubuntu:~/Documents/University-Dissertation$ python3 droidloader.py graph
usage: droidloader.py graph [-h] [-t {permissions,classes}] [-d COMPAREDDATA] [-s SIZE] [-c CATEGORIESIZE] targetdata
droidloader.py graph: error: the following arguments are required: targetdata
alex@ubuntu:~/Documents/University-Dissertation$
```