Analysing Android malware and Testing Detection Tools

Jack Gates – 1500763
CMP 403 Honours Project Dissertation
School of Design and Informatics Abertay University, Dundee

A thesis submitted for the degree of
Bachelor of Science with Honours in Ethical Hacking
30th April 2019

ABSTRACT

Android – the leading mobile operating system has had a significant increase in the amount of malware effecting their devices over the past years. Detection tools developed to combat malware had proven to be ineffective at the start of this spike in malware; although, due to the efforts of researchers these tools have drastically been improving over the years. As these detection tools advance, so does Android malware, evolving new techniques and behaviours to combat detection. The purpose of this paper is to gain insight into the current malware landscape affecting Android devices. The paper will therefore touch upon several areas, including understanding current android malware features and behaviours, demonstrating the effectiveness of detection tools, and trying to discover any trends in these areas. Another purpose of this paper is to highlight any weaknesses in detection frameworks, and to make any potential recommendations based on these weaknesses.

To do this, this paper splits the methodology into 3 stages, a research stage, an analysis stage, and a detection stage. The project started with a research stage due to Android malware being such a large topic, it requires a lot of research to make the other stages feasible. The analysis stage used a series of tools to fully analyse 3 different malware families of different types, to get a complete understanding of their features and behaviours. The detection stage tested 15 different malware families, belonging to the same 3 different types in the analysis stage. It also researched state-of-the-art detection systems to understand their strengths and weaknesses. This paper managed to fully analyse 3 different malware families belonging to different types, extracting their features and behaviours. It also showed that many anti-virus scanners were still quite weak, and highlighted certain trends effecting the detectability of malware, such as in obfuscation techniques and discovery date. Researching many accurate models over time, lead to an understanding of the best techniques currently used, and lead to a better Insight into these systems weaknesses, which allowed the paper to highlight areas that would benefit from future work.

Keywords: Android malware, evolution, analysis, detection, anti-virus, evasion techniques

TABLE OF CONTENTS

TABLE OF FIGURES

1 CHAPTER 1: INTRODUCTION

1.1 BACKGROUND AND CONTEXT

1.1.1 Advances in Malware

Smart-phones have quickly become one of the most widely used personal computing device. This is due to advancements in smart-phone technology, which has allowed them to ramify into almost every aspect of human life. One of these advancements being the increasing prevalence of mobile applications. Unfortunately, along with the increasing adoption of smartphones, and the increasing use of mobile applications, there has been an increasing amount of malware affecting these devices. This is because these mobile applications are the perfect vector for distributing malware, due to it being easy to trick users to install seemingly benign applications with malicious functionality. At the forefront of this battle against malware is Android – a mobile operating system developed by google. It is estimated (IDC: The premier global market intelligence company, 2018) that as of 2018 Android holds about 85% of the market share and also holds the vastest number of apps within its app store – Google play – which contains around 3.8 million applications, almost double that of the subsequent largest app store. There is also a lack in the effectiveness of the security methods conducted by all these different application markets as infected applications constantly make their way onto them. These factors make it a prime target for malware developers looking to spread malware.

A reported (SecureIDNews, 2011) 400% increase in Android-based malware from summer 2010 to the beginning of 2011 shows that malware was becoming a major issue for Android devices. The definition of malware is software specifically designed to disrupt, damage, or gain unauthorized access to a computer system or device – usually for the benefit of the malware author. The benefit is normally for monetary reasons, or for stealing sensitive user data. There are several different types of payloads and features that allow malware to achieve this. Many reports, such as (Wei et al., 2017) classify malware into different types and families, based on the different behaviours that they perform. The malware type is the general behaviour performed by the malware, whereas the malware family is a term to describe the grouping of common characteristics and features, including attribution to the same original malware author. The authors of the report categorised a large amount of malware into 10 different malware types and 71 different malware families. There were also variations between the different malware families, resulting in a total of 135 different malware variations. These variations show that over time malware has been evolving to ‘survive’ and evade detection from anti-virus tools. Either the malware authors have added improvements in present features, or additional features. Some of which include the addition of certain anti-analysis techniques, to make analysis and detection of the malware more difficult. A study (Zhou and Jiang, 2012) started in 2011 highlights these evolutions, showing that these characteristics in Android malware changed quite drastically over time, as well as an increasing presence of obfuscation and anti-analysis techniques over this period. Researchers in this area should continuously study these evolutions to gain insight into the current Android malware landscape.

1.1.2 Effectiveness of detection tools

The study (Zhou and Jiang, 2012) also highlighted that mobile anti-virus software was deficient to deal with the growing sophistication in malware, through testing numerous anti-virus products. The results of this investigation were that the mobile security software had a 79.6% and a 20.2% detection rate at best and worst respectively. These results show that there was a great need to develop better anti-virus software for Android devices at the time. Some mobile anti-virus applications are completely useless and do not actually scan for malware, doing nothing more than showing a progress bar increase, as shown by a recent test of anti-virus applications (Av-comparatives.org, 2019). Of 250 tested anti-malware applications, 138 were unable to detect more than 30% of the malware samples used in testing due to them being buggy and poorly implemented. These results show a drastic need to improve the techniques used in detecting malware.

There are various types of detection techniques used against malware as identified in a report by (Sawle and Gadicha, 2014). A few of the main techniques illustrated in this paper are signature-based detection, behaviour-based detection, and specification-based detection. Signature-based detection in its simplest form is simply cross-referencing the contents of a file with the code signatures belonging to malware known by the scanner. If the signature of an application matches the signature within the malwares data-set, then the scanning software deems the application as malicious. Behavioural-based detection focusses on the run-time behaviour of android applications, monitoring things such as system calls, and the scanner deems an application malicious if it performs any malicious functionality. The article describes specification-based malware detection as being a rule-based detection method that relies on applying rules of what the scanner considers normal to applications. Depending on the system, if the application breaks a certain number of rules the scanner will classify it as malicious or risky. The definition of these categories of detection types are quite loose, and there may be different interpretations of each method. Detection methods also overlap in some areas. For example, (Zhao et al, 2012) combined methods by developing a behaviour signature-based malware detection framework named ‘RobotDroid’ to obtain the signatures of software instances by inserting monitoring points in the service management of Android.

There is arguably an over-reliance of signature-based detection among these anti-virus tools. A report by (Zhou and Jiang, 2012) gave insight into detection frameworks and mentions that traditional signature-based approaches have been demonstrably not promising at all. This is because although it is simple, signature-based detection is very limited as it is only reliable against known malicious samples. This sort of detection disregards any zero-day threats that might appear, of which there are many. Due to this reason, when researching detection systems, this report will look at the approaches in systems that do not rely heavily on signatures.

1.2 AIMS

Overall, this project has two aims: to gain insight into the current Android malware landscape; and to recommend improvements for detection frameworks. Achieving the first

aim means that this paper provides knowledge into the different analysed malware families, explaining their features and characteristics. It also means sufficiently testing the detectability of different malware types, showing any patterns or trends, and illustrating the effectiveness of the scanners themselves. Achieving the second aim means that this paper discusses weaknesses in any of the areas surrounding the detection of malware, and that the paper provides any recommendations of future work, or possible solutions to these weaknesses.

1.2.1 objectives

The first objective is to analyse specific malware families using tools to understand the malware's behaviours and characteristics. This will demonstrate the kinds of tools that are available for analysing malware, the different kinds of features and characteristics present in these malware families, how these extracted features may be useful for classifying malicious behaviours, as well as an understanding of the difficulties presented from anti-analysis techniques. Analysis of different malware types will ultimately give a deeper insight into the current Android malware landscape – highlighting the capabilities of these specific types. The next objective is to test the detectability for each of the malware types. This is to demonstrate the current capabilities of anti-malware tools, and highlight any trends. The final objective is to recommend improvements for these anti-virus tools. Understanding the strengths and weaknesses of a range of different anti-virus tools will help achieve this.

1.2.2 Research questions

This paper will answer the following questions:

- What malicious behaviours, characteristics, and evasion techniques do the chosen malware families possess?
- How effective were the anti-virus tools at detecting these malware samples?
- What trends in detection, if any, are present in the detection results?
- what improvements to anti-virus tools could result in better detections?

1.3 STRUCTURE

Chapter 2 will discuss work related to this research, as well as give more context into the different areas surrounding the objectives. These areas include malicious behaviours present in malware, analysis techniques, and techniques employed by malware to evade detection. Following this, chapter 3 will go over the methodology followed for each of the different stages in this investigation. These stages in the following order are a research stage, an analysis stage, and a detection stage. The purpose of the analysis stage is to gain an understanding of the different areas surrounding malware analysis and detection – this will help prepare for the following stages. The analysis stage involves using tools to analyse different malware types, and the detection stage involves testing anti-virus tools as well as research improvements. Chapter 4 will detail the results of the findings in stage 2 and 3. Chapter 5 will evaluate any findings and discuss various topics involved in this report and Chapter 6 will highlight any conclusions.

2 CHAPTER 2: LITERATURE REVIEW

With Android OS being commonly targeted by malware authors, it is vital that the detection of this malware is as effective as possible to secure its users. To give context, this chapter will start off by discussing older studies performed into the state of detection frameworks. It will then demonstrate the current malware landscape, discussing behaviours and analysis techniques, as well as how malware evades analysis and detection. It will summarise by giving insight into the current state of detection frameworks.

2.1 MALICIOUS BEHAVIOURS

To be able to analyse malware effectively, the analyser should understand malicious behaviours and characteristics well. Malware authors have many motives as identified in a report by (Sawle and Gadicha, 2014) who describe these numerous motives. The two largest motives are making a profit, and stealing personal information. The different malware types will usually focus on one of these motives, for example spy-ware will steal personal information, whereas a banking trojan will focus on stealing or scamming money from the victim. The report by (Zhou and Jiang, 2012) comprehensively covers many different characteristics. The first characteristic that the report delves into is the different methods that the malware uses to install itself on the user's device. The report identifies three methods:

1. Repackaging is the name given to a downloaded benign application which a malicious author disassembles, and then re-assembled with a malicious payload inside.
2. Update attack is an attack which does not enclose the payload within the malicious application itself. Instead, enclosed within an application is an update-component which will fetch the payload. Repackaged applications could also contain an update-attack.
3. Typically, malware uses drive-by download attacks as a means of tricking users into downloading malicious software from a website.

The report also classifies the type of payload as a characteristic. The four types of payload that it lists are:

1. Privilege Escalation which will attempt to bypass the restrictions imposed by the Dalvik VM's sandbox environment and root a devices OS by exploiting known vulnerabilities, gaining control of it undetected.
2. Remote Control, which after installation the bot will communicate with a malicious C&C server which will control the infected device.
3. Financial Charge will find ways to charge the user without their knowledge. An example of this is SMS-trojans, which will automatically send SMS messages to premium-rate numbers, charging the user a fortune.
4. Information Stealing which collects data quietly in the background, and sends this information to a 3rd party without the user's knowledge.

As mentioned in the introduction, classification of malware into different types depends on their features and characteristics. However, the article (Ismail et al., 2017) shows there is quite a lot of overlapping between the different types of malware, making it hard to taxonomize. As an example, two different types of malware listed within the source are a 'bot' and 'trojan', which the article describes in detail. To summarise, a bot is when a C&C server controls the device, and a trojan is an application which appears benign but carries out malicious activities. Some trojans have been known to incorporate bot activity, meaning that malware analysts may class it as both a bot and a trojan. These defined types are therefore too general to classify specific Android malware based on its behaviour. A paper by (Wei et al., 2017) does a better job at categorising malware based on its type by giving them more appropriate names. The different types of malware chosen to for analysis in this project will be based on the types defined in this report, more details on this is in chapter 3.

2.2 Analysis Techniques

There are different methods for analysing malware. Static analysis is the name given to the method which searches through only the applications code. Analysis of the application at run-time is known as dynamic analysis. Dynamic analysis is quite effective at successfully detecting transformation techniques, although it can sometimes be quite resource intensive which can cause some issues. On the other hand, Static analysis poses the advantage of being simpler and more efficient when it comes to data extraction, so it is better suited on memory-limited Android devices. However, its limitations involve being poor at handling advanced transformation techniques. Analysis techniques can either utilise a static or a dynamic approach to analysis, although some systems such as the system by (Salem, 2018) utilise both so that both approaches can work together and mitigate each other's weaknesses.

To understand the different types of behaviours that are present in an application, a malware analyst can use different techniques. Analysing these different behaviours allow for a better understanding of an application and helps determine whether it may be malicious or benign, and to potentially categories it as belonging to a certain malware family. A paper by (Tam et al., 2017) shows that there are many different techniques for Android malware analysis. One of the approaches described is through network traffic analysis. Malware performs network connections to retrieve commands or send data collected from the device to potentially malicious C&C servers. Recording this data going to and from an Android device can uncover data leakages, or unknown downloads. An article by (Shabtai et al., 2014) presented a system which detects meaningful deviations in a mobile application's network traffic pattern, which it did with accuracy. The system was effective at detecting malicious deviations and able to successfully detect an emerging malware with self-updating capabilities, showing that monitoring network traffic may be promising for detection frameworks.

Monitoring API's can also be an effective method of analysis. API's are a set of routines, protocols, application libraries, and tools used for building software applications. Android's use of permissions does restrict access to several critical API calls. A program

could be malicious if it attempts to use restricted API calls when the manifest has not requested the permission, as the application could be trying to leverage a root exploit. Use of certain API's commonly found in malware could also arouse suspicion in an application.

It could be useful to look at data dependencies or control dependencies between different components of an application. An example of a data dependency could be a variable that is dependent on the value of some other event, and a control dependency is the opposite. Malware analysts sometimes compare dependency graphs for similarities such as plagiarism, finding malicious signatures, or for detecting information leaks.

Inter-process communication analysis is based on analysing the ways in which different threads across process boundaries communicate. Inter-process communication and remote procedure calls work together to carry out most tasks within the Android operating system. These remote procedure calls can happen because Android's kernel comes with a binder framework. This inter-process communication only occurs through the binder driver in the Linux kernel. Inter-process communication ultimately allows applications to communicate with remote object methods the same way they would with local methods. Analysing these communications could give useful information regarding information flows, and potential data leaks.

Malware analysts can statically analyse hardware usage through the Android Manifest file as it lists all the hardware components. Some of the different combinations of these hardware components could imply that the application may not be what it claims to be. An example of this would be a calculator application that requires GPS access and a network connection. Malware can also abuse these hardware components. The depletion of battery life is becoming a popular malicious behaviour on Android devices, mainly because it is hard to detect, as detection frameworks rarely look at the hardware components listed in the manifest, and because there are different ways to deliver this payload. Two popular methods are Denial of Service attacks and the installation of GIFs.

Malware analysists perform taint analysis to view the flow of information, moved or copied to new locations. It does this by 'Tainting' data from untrustworthy sources or user data by inserting a tag used to track the object or data during run-time. There are different propagation rules that taint analysis can follow, such as if a tainted object comes in to contact with an untainted object, the untainted object becomes tagged also. This type of analysis is usually for detecting information leaks within an application.

Other methods of analysis include permission usage. An explosion of undesirable behaviours in Android applications resulted from how the platform grants permissions to apps. Some apps request to have permissions that they probably should not need, and it may not seem important to grant them this permission. However, when a user gives an app permissions' after installation, the app could then use these permissions with no further restrictions. The malware characterization article by (Zhou and Jiang, 2012) highlights the differences between permissions requested in benign and malicious applications. Analysing the requested permissions could therefore be very telling of how an application may behave. However, looking for suspicious permissions is not an effective way of analysing

malware that makes use of Root exploits, since root exploits elevate permissions through other means. In a report, (Muñoz et al., 2015) found that the meta-data of an application available on Google were useful in analysing applications. However, there are limitations to meta-data analysis as a lot of malicious applications are from 3rd party application stores. In the report, the authors gathered many applications' meta-data from Google Play, then using a machine learning approach, a system manages to identify the highest predictive meta-data features associated with malware. The study found that information regarding certificate and developer, as well as intrinsic application information are the most promising candidate groups for the detection of malware using meta-data, along with the application category.

2.3 How Malware Evades Analysis and Detection

Because of the poor performance of anti-virus tools, Malware analysts were starting to take the techniques that malware may perform to evade or obstruct analysis and detection more seriously following the investigations into detection frameworks. This prompted more investigations into these malicious techniques.

Anti-analysis and obfuscation techniques were starting to become more common in Android malware after the massive spike in growth around 2011. To test how easy and effective these techniques were, a project by (Rastogi, Chen and Jiang, 2013) involved performing several different transformation techniques on malware samples. A series of anti-virus products then scanned the transformed applications to test how effective they are at avoiding detection. The main findings of the investigation were that transformations on malware samples were generally easy to perform and that all the tested anti-virus products were vulnerable to common transformations. This also makes analysis of these applications to understand their behaviour much more difficult. A report by (Harrison, 2015) corroborates this by investigating the effects of obfuscation on the reverse engineering of Android applications. The author concluded that even basic obfuscation techniques can significantly increase the difficulty and effort required to understand the behaviour of the obfuscated application, showing that these techniques are still effective.

There are several different types of Obfuscation and anti-analysis techniques that android malware may utilise as shown by (Tam et al., 2017). A study done by (Pomilia, 2016) specifically on obfuscation techniques explains these techniques in detail and categorises the different types of obfuscation techniques by how severely they impact analysis. Some of these techniques are:

- Changing of APK data
- Control flow obfuscation
- Loop unrolling
- arbitrary code injections
- Function Inlining/outlining
- interweaving classes
- Anti- debugging
- Multi-threading

- bytecode encryption
- Java reflection
- dynamic code loading
- native exploits

These reports also briefly go over other anti-analysis methods affecting the dynamic analysis of malware. This includes sandbox detection, app collusion, UI intensive applications, Timed execution of malicious code and Require presence of other applications. However, they do not go into as much detail. An article by (Blasco et al., 2018) detailed app collusion which is when multiple applications cooperate to perform malicious behaviours, which is especially difficult to detect due to anti-virus tools scanning applications individually and the applications seeming benign on its own.

There are also methods used by android malware to evade dynamic analysis as detailed in a paper by (Vidas and Christin, 2014). Dynamic analysis is a method of analysis that is becoming increasingly popular, but is still flawed as demonstrated in this paper. The technique focussed on in this paper is known as sandbox detection. If the malware detects that it is running within a virtual or emulated environment for testing then it will not run any of its malicious behaviours and dynamic analysis will not gather any useful data. This paper shows that there are many different methods that could allow for detection of an emulated environment, ultimately, showing that analysis using emulators is flawed. The paper also mentions that presently there are no methods existing to counteract this behaviour. Other anti-analysis techniques used by malware include: timed execution of malicious code; Requiring the presence of other applications, and making activation of malicious functionality very UI intensive.

The research stage of this project involved investigating all these obfuscation and anti-analysis techniques, as well as any potential mitigations against these techniques. This was in the case that they became an obstacle while analysing the malware. Noted were any of the tools involved in these mitigations. Appendix ‘A’ contains an excerpt from the created paper on obfuscation techniques, used to aid the analysis stage, offering a full description of each technique.

3 CHAPTER 3: METHODOLOGY

The methodology outlines the process followed to gain insight into the current state of Android malware and detection frameworks, as well as find areas of improvements within the detection frameworks. It starts off by detailing the different areas researched to gain a general understanding of the area, and to prepare for when it comes to analysing malware. The analysis stage discusses the tools used to analyse malware, as well as a brief account of the challenges faced when using these tools against the chosen malware samples. The detection section explains how the project measured the effectiveness of different anti-virus tools, as well as discusses how this project researched and evaluated potential improvements of anti-virus products.

3.1 RESEARCH STAGE

Malware analysis is a broad subject with many areas needing investigated to have a fuller understanding of the area, so a research stage would be necessary to better understand the basic theory behind malicious Android applications. Certain areas researched further in this stage, include:

- Malicious behaviours performed by malware
- Analysis tools, and the techniques they use to detect malicious behaviour
- Malware Obfuscation techniques

This research involved finding online resources and academic papers on the given areas. Using the information gathered from investigating these areas, the criteria used to find appropriate malware in the next stage will be known. This criterion includes different behaviours, techniques any behaviours that would be interesting to analyse. The literature review in chapter 2 details all the useful information gathered throughout the research stage, and Appendix 'B' has an image copy of the chosen criterion.

3.2 ANALYSIS STAGE

For this stage, there will be an in-depth analysis of three selected malware families. The analyst first gathers malware from two datasets - Koodous (Marcelli et al., 2019), and the Android Malware Dataset (Argus Cyber Security Lab, 2019). The analyst configures the selected tools and then analyses the malware. For each of the selected malware, the analyst creates a document which records all the found details and behaviours.

3.2.1 Tools

Throughout the research stage many tools were briefly looked into. The researcher notes any tools of interest during the research into different analysis techniques; As well as during the research into mitigations for anti-analysis techniques.

Before diving into reverse engineering, it is beneficial to understand the kind of anti-analysis or obfuscation techniques an application may have. A tool named APKiD by (rednaga, 2016) allows an analyst to gather creation information about an APK. This information includes any used encryption methods, or if the malware potentially has any

emulator detection. Simple reverse engineering tools for decompiling applications and looking at the code. The analyst can look at readable source code with simple reverse engineering tools, like dex2jar by (pxb1988, 2015) and APKtool by (Wiśniewski and Tumbleson, 2015), for decompiling applications. The APKtool can decode resources to nearly original form and rebuild them after making some modifications. This allows an analyst to scan through all the files used in the APK. Dex2jar is a tool for converting Dalvik executables to a simple jar file. A tool named JD-GUI by (JD-GUI, 2015) allows an analyst to easily read these jar files and navigate through code. This allows for easy navigation through the code – allowing the ability to browse the reconstructed source code for instant access to methods and fields.

Other tools were also utilised, including DroidBox by (pjlantz, 2014). DroidBox is a dynamic analysis tool used to automatically collect information from an application hosted on an emulated environment. After analysis it generates the results, displaying: Hashes for the analysed package; Incoming/outgoing network data; File read and write operations; Started services and loaded classes through the DexClassLoader; Information leaks via the network, file and SMS; Circumvented permissions; Cryptography operations performed using Android API; Listing broadcast receivers; Sent SMS and phone calls. For taint analysis, the analyser used the object and flow sensitive tool named FlowDroid by (Arzt, 2018). This tool statically scans applications for sinks and leaks.

Something that ended up being very useful for this stage are the tools available in the MobSF framework by (Abraham, 2018). MobSF is as an automated, all-in-one mobile application pen-testing framework capable of performing static, dynamic and malware analysis. Using a combination of many tools the framework can extract suspicious permissions, display all information about the decompiled files. List all activities, services, receivers, and providers. It performs Android API analysis; lists all browsable activities; performs a security analysis for the manifest, the code, and any available files it finds; It performs malware analysis and checks; it lists all interesting URLs, Emails, or strings, and all component names from the code. MobSF also uses a variation of the tool DroidBox for dynamic analysis of applications. It has rooted android emulators set up with certain software to make these results as affective as possible. However, the analyser did not end up using a large portion of the noted tools for this investigation. This is due to the several of the noted tools being either not available to download; outdated and not work effectively at analysing malware; give limited results; or just because of configuration failures due to unknown reasons and not enough documentation available to help.

3.3 Detection Stage

3.3.1 Testing detection

To test the detectability of the malware samples the tester will use VirusTotal. VirusTotal is a free service that analyses files for viruses, worms, trojans and other kinds of malicious content. It does this by inspecting the files with over 70 different anti-virus scanners, and

displays the resulting detection of each engine. The only results considered are those from the 60 anti-virus engines that could perform scans on APK. VirusTotal's antivirus engines are not always the same as the public commercial versions. Very often, antivirus companies parametrize their engines specifically for VirusTotal (stronger heuristics, cloud interaction, inclusion of beta signatures, etc.). Therefore, sometimes the antivirus solution in VirusTotal will not behave the same as the equivalent public commercial version of the given product. So, for this reason, this paper will not compare the engines with each other, but will use the results to measure the detectability of the malware, and the overall effectiveness of the tools.

To improve the detection results, the tester will scan several more families. These additional chosen families will be of the same type as the original three – a Trojan-spy, a trojan-dropper, and a trojan-banker. These other families will range in discovery date, as well is any anti-analysis/detection techniques that they utilise. This is to see if there are any trends or patterns within these areas. The tester scans five samples from each family, and the average detections from each anti-virus engine will be the number used to represent the malware family's detectability. The selected families are in the table below.

Trojan-spy	**Trojan-Dropper**	**Trojan-banker**
Triout	VikingHorde	Mysterybot
Mecor	Kemoge	Lokibot
Finspy	Ztorg	SlemBunk
Vmvol	Ramnit	BankBot
SMSzombie	Boqx	Zitmo

Table 1 - malware families

In theory the same sample should yield the same results, however this is not always the case for all VirusTotal's anti-virus engines, so the tester will perform multiple scans for each strain. The tester scans each sample 3 times, and the result of the strain counts as a detection if an engine detects it as malware in only one of these three scans. The tester places the result of each sample in an excel sheet along with the results of the 60 engines tested on. The tester does this for the 3 analysed families, along with the other 9 selected families.

To get a feel for the false positive rate of VirusTotal I decided it would be a good idea to test benign applications as well. I downloaded 20 different applications from the google play store. These 20 applications were of 4 different categories: games, communication, dating, and finance. This is to give variety to the application. From the 20 applications, only one gave a single false positive out of 60 engines. Showing that VirusTotal has a low false positive rate.

3.3.2 Improving anti-virus

Like the research stage, A researcher will look at several different anti-virus detection frameworks to give insight into the overall effectiveness of anti-virus detection framework on Android. The researcher will then create a document which aims to evaluate a range of detection frameworks from mid-2012 to the current year. The different systems that the researcher looks at will utilise different approaches in what features it extracts for its

detection model, such as XML data within the manifest, or perhaps looking at bytecode semantics. The researcher will look at both systems that utilise static and Dynamic approaches. In the document, for each detection framework evaluated, the researcher will describe the overall system. It will also include: details regarding the dataset used for testing/training the system; the method the system follows; a description of how accurate the system is; as well as any limitations effecting the system. A table illustrating the differences among these frameworks is in the results in chapter 4. There is a discussion on any important evaluations in chapter 5.

4 CHAPTER 4: RESULTS

This chapter details the results gathered from the practical elements of this report, from both the analysis stage, and the detection stage. It will start off by describing the malware that the analyst chose for the analysis stage based on the sought-after criteria decided after the research stage. Following this, is a description of the other details and features that the malware possesses. In the detection portion of this, are illustrations of the results of all anti-virus scans, highlighting any trends that may be present among the different samples.

4.1 ANALYSIS RESULTS

For the Analysis stage of this report, the analyst needed to select three malware families. To do this, the analyst used a guideline of certain criteria, created from the research stage. The information gathered in the previous stage helped in the creation of this guideline. The criteria followed includes details such as the age of the malware, the nature of the malware's payload, and optional features that could be interesting to study. The full criteria list is in Appendix 'B'.

The first selected malware is MysteryBot, first discovered in June of 2018. This is a hybrid malware, the main functionality being that it is a Banking-Trojan that steals sensitive data. The next malware is VikingHorde, discovered in May 2016, which is a Trojan-Dropper which downloads unwanted application. It also makes the victims device a bot which receives instructions from a C&C server. The last selected is Triout, which is a Trojan-Spy discovered in May 2018. This malware steals very personal user data and uploads it to a remote server. Some of the selected malware families show signs of obfuscation and anti-analysis techniques.

4.1.1 Obfuscation evolutions

for the analyst to read the code and understand the malicious behaviours, the java byte code of the malware sample is decompiled with dex2jar, and displayed with JD-GUI. However, some of the obfuscation techniques used throughout made it almost impossible to understand the code by looking at it. An example of this is the MysteryBot sample. After the analyst used APKiD on the sample, they discovered that it uses the 'Allatori Demo' Java obfuscator, which according to their website (Allatori.com, 2019), makes reverse engineering nigh on impossible. Some of the features of Allatori are renaming, flow obfuscation, and string encryption. Looking at online resources (Gahr, Phuc and Croese, 2017) also revealed that the malware uses basic anti-emulator detection as well which did not show up in the APKiD results. Due to anti-analysis techniques, the analyst could not read VikingHorde's main malicious payload. This is because the malware had most of its malicious behaviours hidden in an ELF binary file, which would be far too difficult to reverse engineer. To rectify these issues, the analyst looked at online resources for the malware. The final malware, Triout, did not use any kind of anti-analysis techniques. When selecting the other malware families that were of the same type, it appears that there were trends in the obfuscation/anti analysis techniques. Trojan-Spy's typically had little to no anti-analysis techniques present, Trojan-Dropper's had moderate use of anti-analysis techniques and

tended to favour native payloads, and Trojan Bankers appeared to utilise the widest range of obfuscation techniques.

name	Type	Renaming	String encryption	Dynamic Loading	Native payload	Evade dynamic analysis
triout	Trojan-Spy	no	no	no	no	no
mecor	Trojan-Spy	no	no	no	no	no
Finspy	Trojan-Spy	yes	no	no	no	yes
Vmvol	Trojan-Spy	no	no	no	no	no
SMSzombie	Trojan-Spy	no	no	no	no	no
VikingHorde	Trojan-Dropper	no	no	no	yes	no
kemoge	Trojan-Dropper	no	no	Yes	no	no
ztorg	Trojan-Dropper	yes	yes	yes	No	yes
ramnit	Trojan-Dropper	no	no	no	no	no
boqx	Trojan-Dropper	no	no	no	yes	no
Mysterybot	Trojan-Banker	yes	yes	Yes	no	yes
lokibot	Trojan-Banker	yes	yes	Yes	no	yes
SlemBunk	Trojan-Banker	no	yes	yes	yes	no
BankBot	Trojan-Banker	yes	yes	yes	no	yes
zitmo	Trojan-Banker	yes	no	no	no	no

Table 2 - obfuscation techniques

4.1.2 Malware Features

After analysing the VikingHorde malware, the analyst found that its primary malicious functionality is Installing unwanted applications on the device, which is what a trojan-dropper does. Through reverse engineering of the sample, and looking at an online blog (Check Point Software Blog, 2016) to understand the Native payload, the analyst found that the malware had other functionality. On rooted devices the malware installs native payloads to a device's 'root/data' directory. One binary payload is responsible for implementing a communication protocol with the C&C server. Within this communication, the malware sends updated device information to the server. The other is responsible for updating and maintaining the initial payload on rooted devices, making the malware persistent. If the device is not rooted, then the malware loads the binary as a library in a native wrapper. Within this communication the malware sends sensitive device information to the server and updated when changed. According to the blog, the purpose of this malwares botnet is

for generating income through clicking adds. The botnets host hides their IP address behind the infected devices using proxies to bypass the adds anti-fraud mechanisms by using distributed IP addresses. Also mentioned in the blog is that a user review had claimed the botnet had sent premium SMS messages, charging the victim money, although the analyst could not confirm this during analysis.

The main purpose of Triout is to spy on the unsuspecting victim. The malware has a lot of sinister functionality which involves stealing very personal user data. The malware records phone calls, logs text messages, and tracks the victim's location. The malware sends all this data to a C&C server, although it is unknown what the malware authors use this personal data for. The malware has some incomplete functionality, and the authors made no attempt to obfuscate it, suggesting that the malware may be an accidentally leaked work in progress. Since this malware is relatively new and possibly still in production, a limited number of samples are available.

MysteryBot is a hybrid malware with a lot of different functions, although the main purpose of the malware is to perform trojan banking functionality, such as keylogging. It also has functionality which allows it to overlay over targeted applications. The malware requests to be the default SMS manager to allow control over all messages. The malware requests many other permissions but does not fully use them as the malware authors seem to still be developing certain features, such as ransomware functionality, in which the malware attempts to individually place files in a password protected zip and demand ransom. The malwares main purpose is to send sensitive information, such as messages, keystrokes, and other logs to the C&C server.

4.2 Detection Results

4.2.1 Scanners performance

Out of all VirusTotal's scanners, 60 malware scanners that were able to scan APK files. Out of 82 samples ranging from October 2011 to May 2018, consisting of 15 malware families of 3 different types, the tester found the average detection rate of a sample to be around 33/60 detections. Of the three different types of malware, the tester found the most detectable to be Trojan-Droppers, followed by Trojan-Spy's, then Trojan-Bankers as illustrated in the graph below.

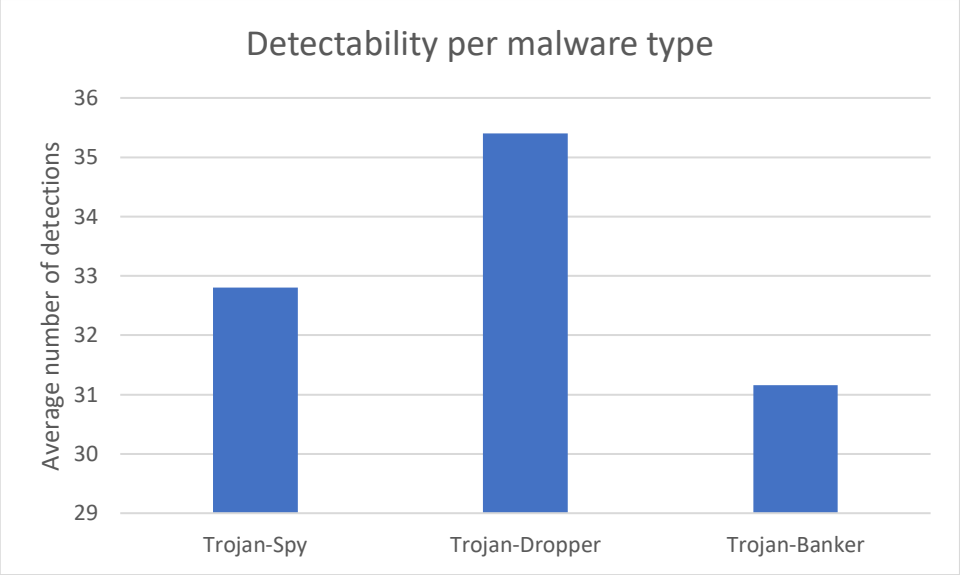

Table 3 - malware type detectability

The 60 scanners varied quite a lot in results between them. Some were able to detect all samples and some detected very few samples. Appendix 'C' Displays each VirusTotal scanner along with their number of overall detections. The top 20 best and worst scanners detected 97.7% and 4.1% of malware respectively. The lower score is probably to do with the creators of the original scanner intending its use for Desktop or other platforms other than Android. The paper will therefore not take this score too seriously. On the other hand, the top 40 scanners on average detected 79.1%.

4.2.2 Trends in Detection dates

Table 4 illustrates how the discovery date of malware impacts the detectability of it. The tester measured the average detectability of each malware family, along with its first detection date. Most families fluctuate wildly, however there is a downward trend when looking at the linear direction of each malware type. This shows that typically the newer the discovery date of the malware, the less detectable it is.

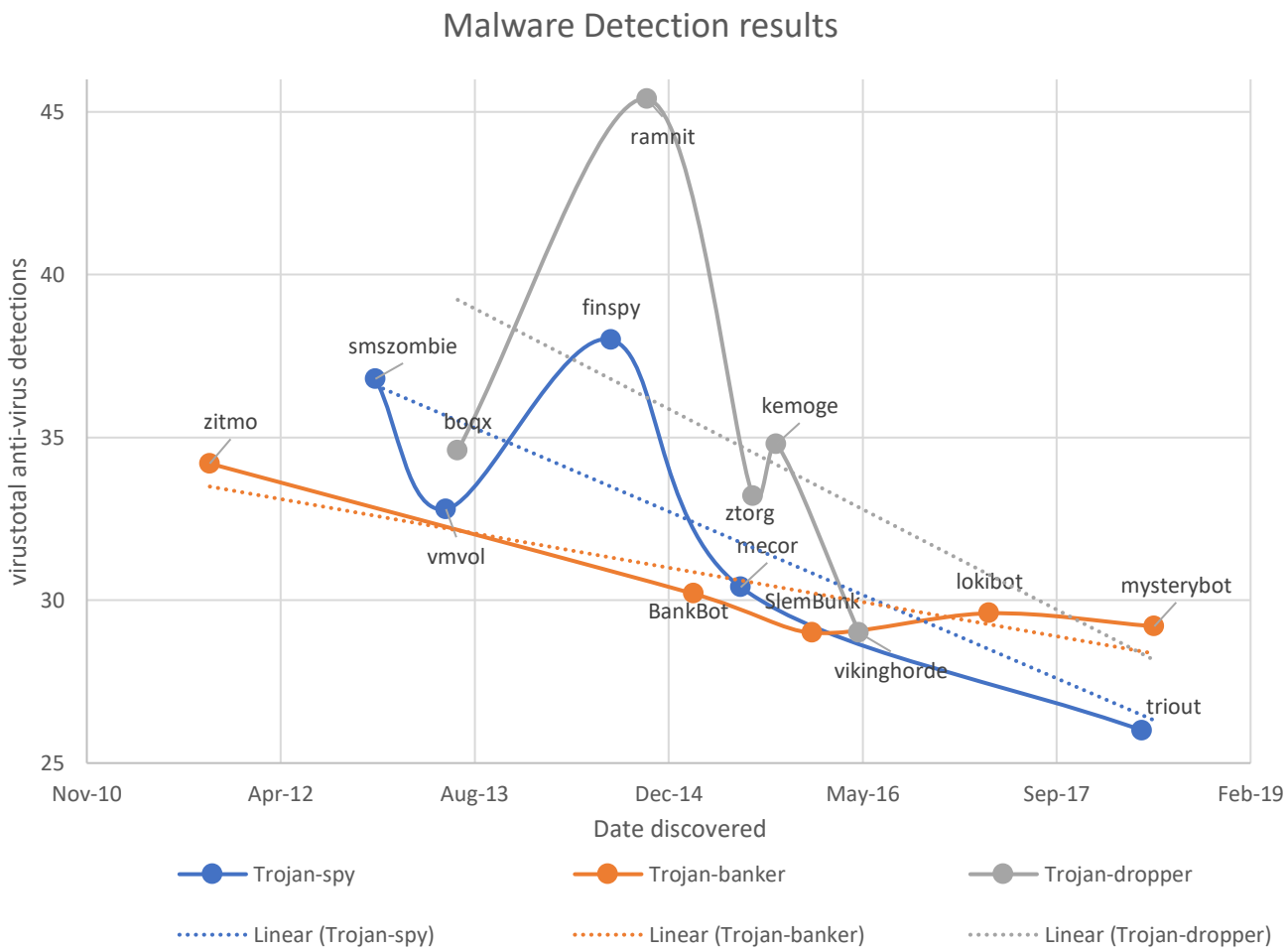

Table 4 - trends in date

4.2.3 Trends in Obfuscation

The tester measured the 5 different anti-analysis techniques effects on the detectability of the malware samples. If a family had one of the following anti-analysis techniques present then the tester would compare its detectability with the average detectability of its malware type calculated earlier. The tester adds the percentage increase/decrease in detectability for each malware family with that specific anti-analysis technique present together and calculates the average. The results for this are rough due to the low sample count. Table 5 shows the average effect each anti-analysis technique has on the detectability of the malware samples. As seen techniques like Renaming, or evading dynamic analysis have quite a little effect. On the other hand, techniques such as string encryption, dynamic loading, or the use of native payloads have a more significant effect.

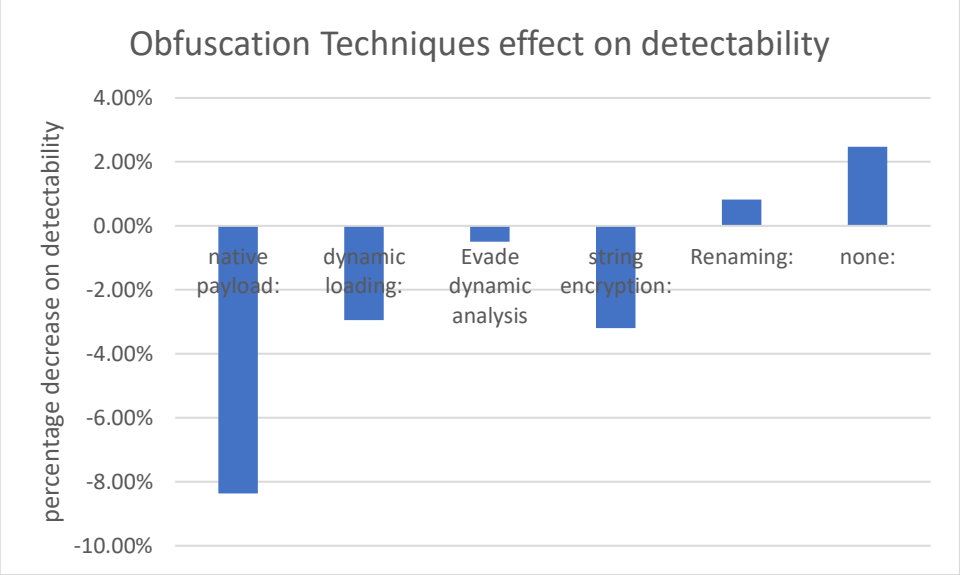

Table 5 - trends in obfuscation techniques

The tester obfuscated the previously un-obfuscated Trojan-Spy samples using a tool called DroidChameleon by (Rastogi, Chen and Jiang, 2013). The tool processed the samples, reversing the order of the code, as well as encrypting the strings present. The 5 processed malware families are in the table below. In four out of five cases, DroidChameleon was able to reduce the number of detections to around 20. This is not the case for SMSzombie, the oldest of its type, which the tool only reduced to 30 detections.

Family	**Obfuscated**	**Un-obfuscated**
Mecor	19/60	32/60
Triout	19/60	25/60
Vmvol	20/60	33/60
Finspy	20/60	38/60
SMSzombie	30/60	35/60
average	21.6/60	32.6/60

Table 6 - DroidChameleaon results

The testing stage incorporated two variants of the same malware into testing. Typically, when malware authors make slight improvements to a sample it gets classed as another variation. The Android Malware Dataset split SlemBunk into 4 different varieties, the first variation used a native payload as its only obfuscation technique, whereas the fourth used native payload, as well as string encryption and dynamic loading. After testing individually, the average detection rate for the first variation is ‘29’, whereas the average detection rate of the fourth variation is ’32.6’. Only in this part of the experiment is the testing results of SlemBunk’s first malware variation used.

4.2.4 Flowdroid

During the analysis stage, the analyst found that the FlowDroid tool was not very effective at finding data leaks within the obfuscated malware. In the completely un-obfuscated Triout malware, the tool discovered that the malware contained 59 data leaks throughout. This contrasts with the two other malware families which had fewer than 10. This prompted a test of this tools ability to test the amount of data leaks on the obfuscated and un-obfuscated versions of these malware families. However, when testing this out the results were quite insignificant.

Family	**Un-Obfuscated**	**Obfuscated**	**Difference**
triout	59	51	-8
Mecor	12	12	0
vmvol	5	5	0
finspy	97	99	+2
zombie	0	0	0

Table 7 - FlowDroid results

4.2.5 Detection frameworks

Table 8 represents all the different detection frameworks that the researcher studied in this investigation. Illustrated in it is the method used by the detection system, the features used for classification, the data required, and a summary of the model's accuracy.

Name	Date	Method	Features	Data required	Accuracy
RiskRanker (Grace et al., 2012)	June 2012	Static	Root exploit signatures, CFG of secretive behaviour, rules applied to the application	bytecode	322 zero-day malware found
Detection of deviations in Mobile Applications Network Behaviour (Chekina et al., 2012)	July 2012	dynamic	Average sent bytes, average received bytes, % of average received bytes, inner average send/received interval, outer average send/received interval	network traffic	94% accuracy but with insufficient testing
Drebin (Arp et al., 2014)	February 2014	static	Manifest features, restricted API calls, used permissions, suspicious API calls, network addresses	XML data, bytecode	93.9% accuracy, 1% FP
An effective behaviour-based Android malware detection system (Zou, Zhang and Lin, 2014)	November 2014	dynamic	Android API logs, function calls to libc, shell commands	API usage, function calls	over 99% with a training set size of 1000

SigPID (Sun et al., 2016)	October 2016	static	requested permissions	XML data	Over 90% accuracy with 22 permissions
Detection of repackaged android malware with code-heterogeneity features (Tian et al., 2017)	August 2017	static	API features, permission usage, features to do with user interaction	bytecode	false negative rate of 0.35% and a false positive rate of 2.97%
ParanoidAndroid (Mendoza and Zhu, 2017)	December 2017	static	Network properties of API usage in a CFG	bytecode	98.7% malware types, 74.1% malware families
DeepRefiner (Xu et al., 2018)	April 2018	static	Manifest features, /res/ features, bytecode instruction semantics	XML data, bytecode	97.74% accuracy
Android Malware Detection based on Factorization Machine (Li et al., 2018)	May 2018	static	Manifest features, restricted API calls, used permissions, suspicious API calls	XML data, bytecode	99.2% detection rate, 0.93% FP, Similar identification rate
Aion (Salem, 2018)	August 2018	hybrid	Manifest features, custom/dangerous permissions, API categories, API calls	XML data, bytecode, API usage	identified 72% repackaged malware
A mobile malware detection method using behaviour features in network traffic (Wang et al., 2019)	May 2019	dynamic	HTTP headers, TCP flows	network traffic	97.89% accuracy

Table 8 - detection systems

5 CHAPTER 5: DISCUSSION

5.1 MALWARE ANALYSIS

5.1.1 features

The malicious applications tended to request a large amount of permissions, many of which were suspicious. Triout requested superuser permissions, giving it the privilege to modify the system at all levels, and MysteryBot requested administrator permissions, granting certain sensitive privileges that benign applications rarely need. Triout also requested many permissions which were telling of its spy behaviour, for example it requires permissions to access camera, record audio, access the devices location and more. MysteryBot also requests permissions that may reveal some of its behaviours. The malware requests the permission to view the currently running tasks, which is a permission required in most malicious overlay modules. The malware uses another permission like this to leak information pertaining to applications in the foreground, however MysteryBot is the first malware to utilise this permission in its advanced overlay module. The malware also requests the permission for disabling keyguard, which is a permission commonly used in ransomware for locking an individual's device. VikingHorde on the other hand, requested far fewer permissions when compared to the other malware samples, and the permissions that it did request were not very telling of its true behaviour.

Through taint analysis of Triout the analyst found a massive number of data leaks throughout the code, where the malware sent a lot of device and personal recorded information over an http connection. It is typical for Trojan-spy malware to leak a lot of personal information, so taint analysis is potentially very useful for analysing these strains. The analyst performed taint analysis on VikingHorde and MysteryBot and found leaks, however they were not of much use in discovering malicious behaviours. In vikingHorde, the analyst recorded 4 suspicious leaks pertaining to leaking mobile details to a rooting application which came bundled with the device, used by the malware to obtain root permissions. The analyst found a limited number of leaks in MysteryBot. This was theorised to be because of the obfuscation techniques used by this malware, although the tester found very limited results. The tester used tools to obfuscate Triout, and other un-obfuscated trojan-spy malware and the results between obfuscated and un-obfuscated malware and found the results to be insignificant, however the taint analysis tool did miss some data leaks.

Other static features in malwares were also useful in identifying potential malicious behaviours. For example, the code disclosed IP addresses belonging to known malicious C&C servers in both, VikingHorde and Triout. When using APKiD the analyst found that several of the malware families used a compiler like 'dexlib', or 'dexmerge' when the authors compiled them. This hints that a malicious separate author repackaged the application because if the compiler used to compile an APK is not the standard Android SDK compiler known as 'dx' but rather a compiler used by reverse engineering tools, then it is more likely someone has modified the application other than the original developers. The intent-filters used by the

malware families can also be quite useful in identifying malicious behaviours. An example of this is in Triout, one class listens for outgoing calls, for the purpose of intercepting and recording it. It was also common to see malware families attempting to hide themselves. This is evident as both Triout and MysteryBot attempt to hide their own icon using the 'setComponentEnabledSetting' method.

5.1.2 Evolution

There were some interesting examples in how the malware has evolved when looking at the samples. An example of how the features of a malware evolve over time are visible when comparing LokiBot to MysteryBot. An article (Threatfabric.com, 2018) describes MysteryBot as being the upgraded version of lokibot due to the two being similar in nature and sharing the same C&C server. There are some additional features in MysteryBot, the first being a new technique used within its overlay module. Malware authors are developing new overlay techniques due to Android versions Android 7 and 8 rendering previous techniques inaccessible. The malware uses the permission 'PACKAGE_USAGE_STATS' to time this attack with precision as it allows the malware to view other applications running in the foreground. This technique used to time overlays affects Android version 7 and 8. The malware has also made changes to its keylogging functionality, although the malicious authors have yet to finish it as the malware does not do anything with the data it gathers. This is unique as analysts have never seen this technique to log key strokes in other malware samples, in essence it calculates the location for each row and places views over the keys in a way that allows the malware to register which keys the victim has pressed. However, the malware relies on the granting the accessibility service after installation for this method to work.

Examples of how the techniques used to allow a malware family to evade or obstruct analysis and detection over time are visible when looking at different variations of SlemBunk samples. An investigation by (Argus Cyber Security Lab, 2019) first discovered the different variations among malware samples, and split them into 4 categories. The earliest variation hides some of its functionality in a native payload, whereas the latest version does this as well as several other techniques. In the latest variation, the malware is more of a Trojan-dropper initially. The application has all its strings encrypted, and the only class is a downloader which dynamically loads and installs the original SlemBunk application onto the user's device. The earliest and latest variations of both malware samples were both tested in terms of detectability in chapter 4, and the tester found that the latest sample was less detectable, showing that these techniques were effective. Another example which had 8 variations, which was the most variations out of all tested families, was a trojan-Banker named BankBot. It started off using no obfuscation techniques, then over time started incorporating simple ones, like renaming, then more complex ones like dynamic loading and string encryption. These specific examples show that even over a short period of time malware, evolves its features and the techniques it uses to become more sophisticated.

5.2 Testing Detection

Overall, the test showed that some scanners detected all malware samples, and many had high detection rates. The top 40 scanners on average detected 79.1% of malware, showing that there is still much room for improvement in many of these scanners.

The differences in detectability of each of the chosen malware samples did not differ too greatly. The malware sample with the least amount of detection did happen to be the Triout sample. This is surprising considering it did not utilise any sort of anti-analysis techniques, whereas the others did. The limited number of detections could however be to do with other reasons, such as it being the newest discovered family out of the three, or perhaps its behaviour is harder to classify as being obviously malicious. Due to the vast number of different variables in each malware family analysed, it is impossible to say exactly why Triout was least detectable. However, other trends in detection are visible when scanning other malware families. These trends were the effect that discovery date had on the malware's detectability, and the effect that obfuscation techniques had on a malware's detectability.

5.2.1 Detection date

In all three of the different malware types tested, as seen in table 4, the detectability of each sample was trending downwards as the families became newer. This strongly suggests that malware gets easier to detect the older it gets, which is likely to do with the scanners increased exposure to older families. The behaviours performed by the different malware families of the same type are typically quite similar, so it may seem unusual that there is a difference based on the sample's discovery date. This is probably in part to do with many commercial scanners using a signature-based detection method, as this method uses databases that constantly need updated to be effective against newer samples.

5.2.2 Evasion techniques

The effects of certain anti-analysis techniques are in table 5. Several had no effect on how detectable the malware was but others were. For example, the malware that evaded dynamic analysis had an insignificant effect on how detectable the malware was as expected due to the scanners using static methods. Common obfuscation methods used by malware such as string encryption seemed to influence how detectable the malware was. This is due to the certain obfuscated behaviours being outside of the scanners reach. The report by (Rastogi, Chen and Jiang, 2013) backs this idea up, by demonstrating that all the evaluated anti-virus products at the time were susceptible to common evasion techniques, string encryption being one of them. The results shown in table 5 here show that several still do. Dynamically monitoring an applications behaviour is one method of tackling these obfuscation methods, however this is not possible for simple anti-virus scanners, although many of these engines probably implement some sort of dynamic monitoring engine in its real version.

Native code seemingly has the strongest effect on the detectability of malware, however only three of the scanned families utilise native code, so this is likely not entirely accurate. Use of native code likely has a strong effect on the detectability of the malware as

the application may have limited malicious components originally, but installs additional malicious components at run-time, therefore the static scanners are missing this functionality. On the bright side there are systems that can easily flag applications that make use of native code or dynamic loading, as their APIs and usages are well-defined. Detection systems, such as RiskRanker by (Grace et al., 2012) use this in its detection of high-risk applications. The use of native code in applications is a real issue when it also comes to anti-virus products due to the way the Android OS architecture is. Due to each application being in its own sandbox environment, complete filesystem monitoring is not possible, as user installed apps do not allow necessary techniques such as hooking. This causes problems when malware like VikingHorde, uses native code to keep itself persistently installed on a device by installing binaries in the root directory. Malware authors do not just use native code for hiding malicious activity, as it is also necessary for any malware which leverages root exploits, which allow attackers to assume full control over a device. According to a report by (Fedler, Schutte and Kulicke, 2013) only 5% of benign applications require the need to use native code. Due to this, there should be more restrictions put in place on the Android system. This report proposes an approach that could limit the abuse of native code through native code hash and signature validation using a whitelist. In this method a private key from the platform provider would sign native code binaries and libraries, and at runtime, a whitelist of benign binaries or libraries would check the applications integrity. This unfortunately means that the platform would not accept many 3rd party libraries, but it would limit the amount of root exploits as well as the hiding of malicious activities in native code.

5.3 DETECTION FRAMEWORKS

5.3.1 Classifying malware

Most of the Android malware detection systems looked at in this paper adopt a very similar process when it comes to malware detection. They start off by gathering a dataset of malicious and benign applications. Afterwards, the system then analyses the applications to extract certain features, which the system processes further to represent each application in the form of a feature vector. A feature vector represents each application, along with a label which is either malicious or benign. Afterwards, machine learning algorithm trains a classifier with this information. There are two separate data sets, a training set, and a testing set. The training set trains the classifier to correctly identify malicious applications, and the testing set tests the accuracy of the system on correctly predicting the applications label. An exception to this process is the system 'Aion' by (Salem, 2018), which performs a more iterative process on malware classification called active learning. In active learning, the wrong classification of an application results in revisiting previous stages in search of a better feature vector for classification. Another exception to this process is RiskRanker, which does not incorporate any machine learning. Instead it measures the risk of applications based on its patterns of behaviour or compliance to certain rules. However, this system focusses less on detection and more on measuring risks in an application for the identification of new malicious malware families.

Many popular features used for classification of malware are in the manifest.xml file. Out of all the systems discussed, the different features used in the manifest were: required hardware components, requested permissions, Different application components, and filtered intents. Five of the studied systems incorporate features of the manifest file in its detection model. Quite possibly the most valuable feature used in the Manifest file is the use of permissions. A system called 'SIGPID' by (Sun et al., 2016) uses only the requested permissions in the manifest file for its model. As mentioned by (Zhou and Jiang, 2012), there is a significant difference in the permissions requested by benign and malicious applications and this system proves that permissions alone are enough to accurately classify malware. However, it also shows that classification of malware based on permission usage alone can result in a high false positive rate. This suggests that although permissions are a very useful feature in classifying malware, alone they can be insufficient. The other studied systems that use features in the manifest.xml file for classification also incorporate other features in its model.

Other features commonly used for classification in these systems are features relating to the Android API. Five of the systems discussed incorporated these features into their detection model, in several different ways. Systems like Drebin by (Arp et al., 2014) and a system by (Li et al., 2018) statically searched through the bytecode for restricted API's. This is because If an application uses an API but does not request the permission to use it, then it is likely that the application is leveraging a root exploit, which is what this system searches for. In a paper by (Zou, Zhang and Lin, 2014) a system dynamically analyses API usage using hooking. This system also intercepts function calls to libc which could give good insight into what the program is doing, which allows it to detect malicious behaviours. All systems using of API characteristics or function calls in their feature vector have very high accuracy rates. However, it is unknown how much this is to do with these features, as they often incorporate other features in the system also.

Network traffic and information was quite common among systems using a dynamic analysis method. Even some static systems make use of network information, for example Drebin incorporated network addresses found in the source code in its detection. However, features relating to network traffic require dynamic analysis for extraction. One system by (Chekina et al., 2012) focussed on the detection of malware through deviations in network traffic behaviour. The system used features relating to the TCP flows, such as the sending and receiving of bytes, as well as the time intervals between them to understand the typical behaviour of an application used by a certain user. Although this method recorded a high accuracy, the authors did not test the system among a large enough sample size or user base to deem how effective it truly was. More recently, a system by (Wang et al., 2019) demonstrated another non user specific way to analyse network traffic. This system does this through using features relating to HTTP request headers and TCP flows. This model showed that the features extracted from Http headers were more accurate at detecting malware than TCP flows, but combined they were even more effective. However, the latter system highlighted that if there are no available training samples for a specific malware family, then it is difficult for this system to identify it. This means that although quite accurate, traffic analysis may not be very good at finding zero-day malware.

The reason that so many detection systems use machine learning is because it allows for the automatic crafting of detection patterns. Automation means that the system handles large volumes of data relatively quickly, saving a lot of time when creating an accurate model. It can also learn behaviours over time that regular detection systems cannot, for example the system by (Chekina et al., 2012) learned the specific user's behaviour and incorporated that into its model for detecting anomalous behaviours. The patterns created by machine learning are also superior to signatures for detecting zero-day malware. However, there are some disadvantages when using machine learning in detection models. One of them being that the model relies entirely on the quality of the application dataset. If the malicious applications used in the dataset do not represent the current characteristics of behaviour of malware then the model will be quite ineffective. Since malware evolves, it is important for these models to be constantly re-trained using an up-to-date dataset. However, creating these up-to-date datasets is not easy for analysts to achieve as discovering zero-day malware families and categorising malware presents its own challenges as discovered in the the section below. Another disadvantage of using machine learning in a detection model is that they usually require automatic feature engineering, generating detection features that are not always straightforward for humans to understand.

5.3.2 Other malware detection types

It is difficult to compare the systems in terms of overall detection accuracy as many of them have different detection goals, rather than simply labelling applications as either malicious or benign. For example, RiskRanker specialises in the detection of zero-day malware. ParanoidAndroid by (Mendoza and Zhu, 2017) specialises in the categorising of malware into different types and families. The system Aion and another system by (Tian et al., 2017) specialise in the detection of repackaged applications. Although machine learning could be very effective at discovering zero-day malware, discovering new families of zero-day malware could be quite difficult. This is due to the feature vectors only consisting of features present in the malware in its dataset, and not new malicious features that are only present in these zero-day families. RiskRanker can effectively find zero-day malware through assessing the risk an application possesses. It measures this risk based on the compliance of the application to a certain set of rules that RiskRanker established as being effective in identifying malware. In the first order analysis it traverses the control flow of the application to find secretive background behaviours that involves sensitive actions, such as financial charge. The second order analysis specialises in identifying anti-analysis techniques common in malware, such as the use of encryption techniques and native code execution. This system examined many applications from various markets, and reported a small fraction of them as risky. After further manually analysing these risky applications around a quarter were malware, half of which were zero-day malware. Although manual analysis is time consuming, it is necessary for verifying that these unknown applications are malicious. For being the first large scale system designed for identifying zero-day malware it was very successful as it discovered 11 new malware families. Adding this malware to datasets could help train future machine learning models, as they will creating more accurate malicious feature vectors that represent a wider range of malicious features for training.

The categorising of malware types and malware families is quite important as it is necessary to have behaviour profiles for malware to understand how they can infect the devices they are on, the threat level they pose, and how to protect against them. The ability to identify and categorise malware as belonging to a certain type, or family is therefore useful in detection frameworks for this reason. One of the models that focussed purely on the identification of Android malware into types, and families is ParanoidAndroid. This uses machine learning techniques on the network properties of API usage for identification. The system proved that this method was effective at correctly categorise 98.7% of malware types. However, it was only able to correctly categories 74.1% of malware families. Although, there were misclassifications in the malware's family, the wrong selected family belonged to the same malware type. This shows that classification of malware types is effective but classifying certain malware families using this system is quite limited. The detection phase of this project illustrates the same problem. Some of the anti-virus scanners identified that the malware was malicious, however they got the family it belonged to wrong, albeit it belonged to the same type. A more accurate method by (Li et al., 2018) was able to correctly classify 98.94% of malware families using a factorisation machine classification model instead. However, its improvements in accuracy could be less to do with the classification technique and more to do with the features it uses, as it incorporates a lot more features such as features from the manifest in its identification.

Repackaging benign applications with malicious payloads is increasingly common among malware authors, and due to this the ability to detect whether a malicious author has repackaged an application or not could be a very useful. The system Aion focussed on the detection of malware using active learning on extracted features. The system uses static features, such as those found in the manifest and any API features found in the code, as well as dynamically features. The system does this by running the application on an Android device and stimulated with a UI-based tool to reconstruct the applications behaviour. The classifier then implements active learning. When detecting repackaged malware, this means that gathered feature vectors leading to a wrong classification will result in the system running the feature extraction process again for the same application in hopes that the UI-based tool extracts features more representative of the applications malicious behaviour. Unfortunately, the detection rate was only 72%, but as mentioned by the authors, it could be do with the UI-based tool used for stimulation being ineffective at executing the correct paths. A paper proposes a system by (Tian et al., 2017) which specialises in the detection of repackaged malware. The authors managed to create a system with high detection rates using machine learning on code-heterogeneity features. The system creates dependency graphs for both classes and methods, which helps partitions an applications code into different subsets. The features extracted from these subsets are sensitive API features, permission usage, and features to do with user interaction. One of the disadvantages of this system though is that the features are based on static code analysis, meaning advanced anti-analysis techniques such as code obfuscation, and dynamic loading could result in poor results.

These systems all show that for the overall detection of malware, these researchers should not just place effort into creating machine learning systems with high accuracy rates

on classification, but also into systems that specialise in other areas, such as categorising malware, or discovering new families. Arguably there is more research needed in these areas, which will subsequently improve the accuracy of these systems using machine learning for classification.

5.3.3 Areas of improvement

In general, improvements in detection systems over time are evident from this research. Although, as explained earlier accuracy may not be the best metric for showing these improvements between most systems, in some cases it is. For example, the system Drebin and the system by (Li et al., 2018) worked on the same dataset using similar static feature vectors, but a different classification model. Drebin achieved an accuracy on this dataset of 93.9% and a false positive rate of 1%, whereas the more recent system achieved a detection accuracy of over 99% and a false positive rate of 0.09%.

The two systems that make use of network properties in their detection are also comparable. One system by (Chekina et al., 2012) models itself around the TCP flows generated from the way a user interacts with an application and classifies a certain level of deviation as malicious, resulting in an accuracy of 94% with no false positives. The more recent system by (Wang et al., 2019) uses other network features as well as TCP flows, such as the features found inside HTTP headers. This system achieved an overall accuracy of 97.89%, however, it has a very high false positive rate of over 10%. This could be because unlike the first system, it does not take specific user interactions with the applications into consideration and generates the traffic automatically. This is beneficial in that it is easier to train the system accurately on a larger set of data, yet also detrimental due to it not tailored to specific users' behaviours. This allows for easier testing, resulting in a more accurate representation of accuracy and false positive rates for the system. Arguably the first system was lacking in test data, and may be less effective than it appears if tested against a broader range of malware and users although this would be difficult and time consuming. To conclude, both systems seem to highlight the challenges faced by systems that dynamically extract network traffic features to train classifiers. The paper by (Wang et al., 2019) highlights that researchers have not explored this area enough, and that there are still a lot of valuable traffic features to test in detection systems.

Perhaps an area of improvement is to develop more detection frameworks that do not rely on machine learning on malware datasets, such as RiskRanker, which incorporates rule-based detection into its model. This system proved to be effective at discovering zero-day malware. More systems like this will improve malware datasets and subsequently the quality of feature vectors used to train machine learning classifiers in other systems, improving their accuracy for real-world use. The detection of zero-day malware is of vital importance. In an article (Lueg, 2017) a security company 'G Data' predicted that there will be 3,500,000 new malicious Android files by the end of 2017 and the production of malware on Android shows no signs of slowing down. The Android malware dataset has shown that there are sometimes years between a malware's creation date and its discovery date, suggesting that currently, there are most likely many undiscovered malware families around.

The identification of different malware types and families based on its behaviour was another thing proven to be useful when it came to detection. Systems such as ParanoidAndroid and the system by (Li et al., 2018) had shown that this is possible to do with high accuracy. The miss-identification of malware is quite prevalent in some of the tested anti-virus scanners used in the testing stage. For example, several of the scanners presented the name of the family that they classified the malware as belonging to and were wrong. Although in all these instances, the family were quite similar and were of the same type. Often scanners will not really give much information relating to the malware and will just give it a generic title, such as trojan. Therefore, the accurate identification of different malware types is another area which would benefit from more research.

As seen from examining these detection frameworks, most of the models study the bytecode statically, this can be effective, but is increasingly becoming a struggle due to bytecode encryption – an increasingly popular anti-analysis technique used by malware authors. Bytecode encryption is the process of modifying the java bytecode of an application, making it harder for the machine to understand semantically, and making it almost impossible for a user to read. The dynamic loading of certain functions is also impossible to statically analyse. Several of these papers have pointed out that failing to identify bytecode obfuscation or dynamic loading is just one of the inherent limitations of static detection models and many of the systems of today still extract features from the bytecode. This should not cause too many issues as bytecode features, used in conjunction with other features will yield better results and is common in several of the researched models. Usually XML data, or even some hybrid systems, like Aion, also incorporate dynamic API usage with bytecode features. Utilising extracted bytecode features into the feature vector is acceptable, so long as the system incorporates other features into the feature vector to avoid it failing to classify heavily obfuscated malware. As mentioned by DeepRefiner, which is a multi-layer model, adding a dynamic layer onto their static layers may improve accuracy due to better handling of obfuscated malware.

To summarise, this paper looked at a wide range of detection frameworks, highlighted what trends were consistent over the years, as well as some changes in trends. One of the trends which stayed consistent was the use of machine-learning in detection models. This proved to be effective at finding malicious patterns between features in benign and malicious applications which resulted in high accuracy rates for the system. Static methods are a popular choice in detection frameworks as they were much simpler to implement and tended to yield much higher accuracy rates in their detection as illustrated in the table. Dynamic models tended to be a bit less reliable in their detection. This is possibly because there is less research on dynamic systems as opposed to static systems, as highlighted in the paper (Wang et al., 2019) researchers have yet to explore many of the different dynamically extracted network features. Some of the static systems have however suggested implementing dynamic detection modules into their models in future, as they believe it will improve their accuracy when classifying heavily obfuscated malware. Something that this paper shows is that signature-based detection is not common in any of these detection models. The only model mentioned that used signature-based detection was the RiskRanker model which used signature matching to find root exploit payloads.

Other than that, none of these state-of-the-art detection systems implement signature detection into their model, which makes it a mystery why it is so commonly depended on in commercial anti-virus products.

6 CHAPTER 6: CONCLUSION

6.1 FINAL THOUGHTS

In conclusion, this work has given some insight into the current Android malware landscape, as well as recommended several areas of improvement for detection frameworks. Through the analyses of malware of three different types, this paper has given a deep insight into each of the different types, and the unique behaviours and features belonging to them. This involves a Trojan-Banker named MysteryBot, a Trojan-Spy named Triout, and a Trojan-Dropper named VikingHorde. Through analysing this malware, this paper has presented a great understanding of the current features and behaviours associated with malware of the same type. The paper also found that the features valuable in discovering the malicious activity are different in each type. For MysteryBot, the biggest indicator for suspicion was the surge of risky permissions that it requests, which far exceeded the amount requested by the average benign application. It makes sense that MysteryBot would request this many permissions as Trojan-Banker's typically have a lot of different functionalities relating to stealing bank credentials, such as overlay attacks, ransomware, and keylogging. For Triout, the biggest indicator of malicious activity was the large amount of data leaks found during taint analysis. Analysts expect a large amount of data leaks from a Trojan-Spy, as their purpose is to exfiltrate sensitive data. There was no one feature of VikingHorde that would make it easy to detect, although a combination of multiple features makes it detectable, such as disclosing known malicious IP addresses; making use of native libraries; and super user commands. Three finished documents each fully describe one of the three malware families.

Incorporating many different families belonging to the same malware type in the detection stage, revealed evolutions in features and techniques within the malware samples. An example of feature evolution is in the Banking-Trojan MysteryBot which evolved from LokiBot, improving several components such as its keylogging, and overlay module to be more effective against newer android API's. This paper also highlighted evolutions in the techniques used to obfuscate or evade analysis. An example in this paper is the SlemBunk malware, which the authors over time incorporated additional complicated transformations to. This made analysis more difficult and the testing stage demonstrated that less scanners could detect the newer variants making use of additional obfuscation techniques. This paper highlighted these evolutions as they demonstrate the ways in which malware is increasing in sophistication, which is important insight for those developing frameworks to deal with these sophistications.

Ultimately the detection stage showed that the top 20 scanners are strong and could on average detect 97.7% of the tested malware samples. Although the results showed that many scanners still struggled to detect certain malware samples as this number dropped to 79.1% when taking an average of the top 40 scanners. The detection stage also highlighted trends in both obfuscation techniques, and the detection date of malware in how they affect its detectability. The malware families using certain obfuscation techniques were less detectable on average. The techniques found to have the greatest effect on malware were

string encryption, dynamic loading, and native payloads. The testing stage involved obfuscating the previously un-obfuscated Trojan-Spy malware, which made a significant difference in making it less detectable. This stage showed that detection date had a significant impact on malware, as on average the more recent the detection date the less detections the malware received. Therefore, the detection stage successfully highlighted trends in malware, giving insight into the current state of anti-virus scanners. Although there is arguably insufficient test data to accurately compare the detectability of each obfuscation technique against each other.

This paper evaluated a range of detection frameworks from mid-2012 to the current year that do not rely on signature-based detection. Several of these frameworks focussed on detecting malicious applications as accurately as possible, typically to do this they extract certain features from applications, process them, and feed them into a machine learning algorithm to train it to classify malware. This paper showed that a range of different systems experimented with a range of different features in their detection model. However, the paper highlighted limitations among many of these features. Certain static features, especially those found within the bytecode, are less effective against highly obfuscated malware. Systems using dynamic features yielded poorer accuracy rates, although several of the researched articles mentioned that there are many dynamic features that researchers have not explored. Overall, systems which find patterns between malicious and benign features show a lot more promise than the traditional method of signature-based detection common among most anti-virus tools.

Some of the frameworks researched had different goals other than the creation of an accurate system. Some specialised in the detection of zero-day malware, some specialised in the detection of repackaged malware, and some focussed on categorising malware into different types and families. Analysts will always require systems that focus on the detection of zero-day malware, especially at the rapid rate malicious authors are creating malware, highlighted by G Data earlier. The typical systems which focus on accurately detecting malware are not always very effective at detecting zero-day malware as the zero-day applications are not in the training set of malicious applications. Systems that incorporate rule-based detection to find risky applications are better suited for detecting zero-day malware. There are few systems that do this, however ore systems like this will improve malware datasets and subsequently the quality of traditional systems focussed on accurately classifying datasets. The models in this paper specialising in categorising malware were quite successful, however, chapter 5 discusses how several scanners are weak when it comes to correctly identifying malware, which highlights this as being another area which would benefit from more research. To summarise, the paper highlighted several strengths and weaknesses of detection systems with different goals, using different methods and features, and discussed possible areas of improvement in detail. A finished document describes all detection systems studied in this report.

6.2 FUTURE WORK

There are several areas in this project which would greatly benefit from expansion. The analysis stage has the limitation that it only covers the behaviours present in 3 different types of malware, whereas the Android Malware Dataset identifies 9 different types of malware. Analysing and testing detection on all the different types of Android malware available will give a full insight into the entire area.

The detection stage also had limitations. This stage only tests the detectability of 15 out of 71 malware families in the Android Malware Dataset. Although most of these are of different types to the three chosen for this investigation, Increasing the number of malware families would improve the accuracy of the patterns and trends identified in the results. For example, it could highlight clearer distinctions between the effectiveness of different obfuscation techniques on the detectability of malware.

The method used for detecting malware may not yield the most reliable results. This is because VirusTotal's antivirus engines are not always the same as the public commercial versions. Another way to test the detectability of the malware is to use the commercial products themselves, although this would be far more complicated and time consuming. Realistically it is likely that the number of scanners in the experiments would far less than 60, although the results among these fewer scanners would be more reliable. Performing the same tests again but with a different set up would give better overall results, as well as draw a comparison between VirusTotal scanner versions and their commercial version.

The paper looked at a total of 11 different state-of-the-art detection systems. The systems varied in date, method, features, the data that they require, and their overall accuracy/goal. Looking at a larger number of detection frameworks would however increase the variety. Increasing the number of different scanners could allow for a greater variety in these areas, as well illustrate how these models have evolved better. The frameworks not present in this paper could potentially make use of features not used in any of the models in this paper, or utilise different methods not thought of.

Finally, this paper only looked at improvements for anti-virus tools to secure Android devices against malware, however changes to the platform could also improve the security of the platform. A way to secure the platform against native exploits is briefly discussed in chapter 5 under the heading 'evasion techniques'. There are possibly many more ways secure the platform, although this paper does not touch on any of them. Placing future research into ways to secure the platform against malware, rather than relying just on anti-virus tools, may be of value.

7 APPENDICES

7.1 APPENDIX A – OBFUSCATION TECHNIQUES PAPER (EXCERPT)

Methods affecting Static Analysis

obfuscation of APK data: This is the obfuscation of APK data. This can include changing the names of files, methods, variables, and field identifiers. It could even cover the reordering or encryption of certain elements within the 'dex' files. This method, if done successfully, can prevent most pattern matching methods used by static analysis.

control flow obfuscation: This method transforms the control flow so that when the code is converted from 'dex' files, the number of possible ordering combinations of the basic blocks is too much for the analysis tools to really understand. there are generally two ways of doing this. The first is by changing the linear ordering in which the code blocks are written, the other is by increasing the number of branches and diversions within the methods.

Loop unrolling: The purpose of this technique is to optimise a program; however, it could also be used for obfuscation purposes as unrolling a loop by a given rate affects the readability of the code.

arbitrary code injections: Injecting irrelevant code sequences, or arbitrary variable checks could also make the code a lot less readable for analysts.

Inlining/outlining: function inlining is the breaking down of functions into multiple smaller functions, whereas outlining is the replacing of function calls with the entire function. These both can be used to change up the code and effect the readability.

interweaving classes: This is when android classes are combined by merging their bodies, the parameters and the methods affecting the readability of the code.

Anti- debugging: This involves stripping away debug information like local and parameter variable names, source file names, and source line numbers. Debug data can potentially be useful information for an analyst, so getting rid of it could make an application trickier to analyse.

Multi-threading: This can complicate static analysis as with multi-threaded programs it is much more difficult to follow the interactions and all the potential interleaving's of instructions between different threads.

bytecode encryption: This involves the encryption of all the relevant pieces of the applications bytecode, including functions and variables. Fully encrypting the bytecode could potentially make statically analysing an application futile as the application only decrypts this code at runtime.

Java reflection: by using reflection (a feature that basically allows a program to manipulate its own internal properties), functions can be invoked implicitly and static analysis tools will

have a hard time discovering which functions or methods were invoked, greatly complicating static analysis. Reflection is often combined with bytecode encryption to hide the names of classes or methods targeted by reflected calls.

dynamic code loading: This technique involves loading a library into memory at run-time, meaning that static analysis cannot analyse it. A malware developer could potentially change the behaviours of applications in an unpredictable way while the application is running.

Methods effecting dynamic analysis

sandbox detection: If malware can detect whether the environment that it is set up in is either virtualised or emulated then it will not exhibit any of its malicious characteristics or behaviours and will stay hidden against dynamic analysis. Sandbox detection can potentially detect an emulated environment through the Android API, the network, or through just detecting the underlying emulator.

app collusion: This is when applications cooperate to perform malicious behaviours which is possible because Androids Inter-Process Communication (IPC) mechanism supports and encourages the sharing of resources among applications. This technique is very hard for dynamic analysis to detect as the applications behaviour on its own may seem benign, but dangerous when paired with another application.

UI intensive applications: Code coverage is a major issue effecting dynamic analysis as it can be difficult for an automated system to simulate user interaction with an application. Making the application more UI intensive would mean that an automated system is more likely to fail in executing the malicious code.

Timed execution of malicious code: A simple method of preventing dynamic analysis is through delaying execution of any suspicious functionality for a certain amount of time. Since dynamic analysis only monitors an application for a limited time, executing it later (after analysis is finished) will prevent analysis.

Require presence of other applications: Some applications, to run their malicious payload, require the presence of other applications. For example, a banking trojan which uses UI overlays to perform its attacks may require the presence of a banking application to run, because the attack would be useless otherwise. This also deters dynamic analysis is the testing environment is unlikely to have the required application.

7.2 APPENDIX B – MALWARE CRITERION

DESIRED CRITERIA IN MALWARE STRAINS:

1. This investigation will be focussed on applications with hidden malicious behaviours. The applications will either be stand-alone, or will be re-packaged applications with malicious payloads. Stand-alone applications are on the rise due to the effective anti-repackaging solutions and are also often more sophisticated as the developers made them from scratch and likely put more effort into creating effective malware, so it is likely all applications will be stand-alone.
2. The way in which the malicious payload is activated is not important for this investigation, however scheduled payloads could cause problems for analysis if using dynamic methods.
3. The strains investigated should be relatively new, 2 years old at most.
4. The motivation behind the malicious application is not important for this investigation, whether it is for monetary benefit or data theft, so long as it can be considered malicious in nature.
5. The types of malicious payloads that this investigation is interested in can be seen below, the investigation will attempt to analyse at least 3 of the following payloads within its samples:
    - **Remote Control/Bots**: After installation the bot will communicate with a malicious C&C server which will control the infected device. This can potentially be used to steal sensitive information, log keystrokes, perform denial of service attacks, send spam, or download other malicious payloads.
    - **Privilege Escalation**: After installation this type of malware will attempt to bypass the restrictions imposed by the Dalvik VM's sandbox environment and root a devices OS by exploiting known vulnerabilities, gaining control of it undetected. Often this type of malware will install a back door for the installation of other applications, or can be used to steal data.
    - **Information Stealing**: This type of malware collects data quietly in the background, and sends this information to a 3rd party without the user's knowledge. This is often used for collecting browsing activity for advertisements (adware), however this investigation is more interested in the more nefarious uses, like SpyWare or Banking Trojans. These Banking Trojans have one focus – stealing any sensitive information related to banking. The malware will often check to see what banking applications the user is running, and adjust its behaviour based on that, creating hidden overlays that it can use to record user data, or keylogging the information.
    - **SMS Trojans**: This malware will automatically send SMS messages to premium-rate numbers, charging the user a fortune.
6. Optional behaviours/characteristics that could be interesting to study and give more varied results:
    - Signs of obfuscation: this will give good insight into how effective these obfuscation techniques are against anti-virus tools. It will also give insight into the current effectiveness of dynamic analysis methods as these are predominantly used for detecting obfuscated malware. Although this will present some challenges for the analysis stage.
    - Application is located on the google-play store (could test meta-data analysis, although it is unlikely that the appropriate malware will be found on the google-play store)
    - Application colludes with another application (colluding applications are hard to find, so this is unlikely)
    - Application depletes battery life (Frameworks rarely look at hardware components, so could be challenging to find/analyse).

7.3 APPENDIX B – INDIVIDUAL SCANNER RESULTS

Scanner:	Detections:
ESET-NOD32	82
Ikarus	82
Qihoo-360	82
Symantec Mobile Insight	82
ZoneAlarm	82
CAT-QuickHeal	81
DrWeb	81
F-Secure	81
Sophos AV	81
Trustlook	81
Avira	80
Babable	80
K7GW	80
Avast Mobile Security	79
Fortinet	79
Avast	78
AVG	78

McAfee	78
NANO-Antivirus	78
Tencent	78-
AegisLab	77
Symantec	77
Kaspersky	76
AhnLab-V3	75
Antiy-AVL	74
MAX	73
McAfee-GW-Edition	73
Baidu	64
Comodo	60
Cyren	57
Zillya	53
Microsoft	41
Jiangmin	35
F-Prot	34
Kingsoft	33
ClamAV	22
Zoner	21
TrendMicro-HouseCall	20
TrendMicro	17
Arcabit	10
BitDefender	10-
Emsisoft	10
eScan	10
Gdata	10
Ad-Aware	5
Panda	5
Rising	5
Yandex	5
K7AntiVirus	3
ViRobot	3
VBA32	2
Alibaba	0
ALYac	0
Bkav	0
CMC	0
Malwarebytes	0
SUPERAntiSpyware	0
TACHYON	0
TheHacker	0
VIPRE	0

8 REFERENCES

Zhou, Y. and Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. [ebook] San Francisco: North Carolina State University. Available at: http://www.ieee-security.org/TC/SP2012/papers/4681a095.pdf [Accessed 16 Sep. 2018].

SecureIDNews. (2011). Android malware volume jumps 400% - SecureIDNews. [online] Available at: https://www.secureidnews.com/news-item/Android-malware-volume-jumps-400/ [Accessed 23 Sep. 2018].

Tam, K., Feizollah, A., Anuar, N., Salleh, R. and Cavallaro, L. (2017). The Evolution of Android Malware and Android Analysis Techniques. [ebook] ACM Computing Surveys, pp.vol. 49, no. 4, pp. 1-41. Available at: https://pure.royalholloway.ac.uk/portal/files/28069262/computingsurvey.pdf [Accessed 16 Sep. 2018].

Rastogi, V., Chen, Y. and Jiang, X. (2013). DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks. [ebook] North Carolina: Northwestern University. Available at: http://pages.cs.wisc.edu/~vrastogi/static/papers/rcj13b.pdf [Accessed 1 Oct. 2018].

M. Zhao, T. Zhang, F. Ge, and Z. Yuan, “Robotdroid: A lightweight malware detection framework on smartphones,” Journal of Networks, vol. 7, no. 4, pp. 715–722, 2012

Harrison R. Investigating the effectiveness of obfuscation against Android application reverse engineering, Technical Report RHUL-MA-2015-7, Royal Holloway University of London, Surrey, UK, 2015.

Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. [ebook] Kyoto, Japan: Carnegie Mellon University. Available at: https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf [Accessed 7 Nov. 2018].

Sawle, P. and Gadicha, A. (2014). Analysis of Malware Detection Techniques in Android. International Journal of Computer Science and Mobile Computing, [online] 3(3), pp.176 – 182. Available at: https://pdfs.semanticscholar.org/7f33/9156f47345bd102c9b05f45f9bfe4c182720.pdf [Accessed 14 Nov. 2018].

Ismail, N., Saad, H., Robiah, Y. and Abdollah, M. (2017). General android malware behaviour taxonomy. Defence S and T Technical Bulletin, [online] pp.160-168. Available at: https://www.researchgate.net/publication/319094411_General_android_malware_behaviour_taxonomy [Accessed 14 Nov. 2018].

Zhang, Y., Yang, M., Yang, Z., Gu, G., Ning, P. and Zang, B. (2014). Permission Use Analysis for Vetting Undesirable Behaviors in Android Apps. [ebook] IEEE. Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6876208 [Accessed 1 Oct. 2018].

Blasco, J., Chen, T., Muttik, I. and Roggenbach, M. (2018). Detection of app collusion potential using logic programming. Journal of Network and Computer Applications, 105, pp.88-104.

Shabtai, A., Tenenboim-Chekina, L., Mimran, D., Rokach, L., Shapira, B. and Elovici, Y. (2014). Mobile malware detection through analysis of deviations in application network behavior. [ebook] Beer-Sheva, Israel: Ben-Gurion University of the Negev. Available at: https://ac.els-cdn.com/S0167404814000285/1-s2.0-S0167404814000285-main.pdf?_tid=a218ea7a-7e82-4453-9b8b-7bc261fe95a0&acdnat=1538862945_a6b45dbf3194f7ef23164c6fe1cb09b3 [Accessed 1 Oct. 2018].

Vidal, J., Monge, M. and Villalba, L. (2018). A novel pattern recognition system for detecting Android malware by analyzing suspicious boot sequences. [ebook] Madrid: Universidad Complutense de Madrid (UCM). Available at: http://www.sciencedirect.com [Accessed 1 Oct. 2018].

Muñoz, A., Martín, I., Guzmán, A. and Hernández, J. (2015). Android malware detection from Google Play meta-data: Selection of important features. In: 2015 IEEE Conference on Communications and Network Security (CNS). IEEE.

Allatori.com. (2019). Allatori Java Obfuscator - Professional Java Obfuscation. [online] Available at: http://www.allatori.com/ [Accessed 16 Mar. 2019].

Gahr, W., Phuc, P. and Croese, N. (2017). LokiBot - The first hybrid Android malware. [online] Threatfabric.com. Available at: https://www.threatfabric.com/blogs/lokibot_the_first_hybrid_android_malware.html [Accessed 16 Mar. 2019].

Marcelli, A., Oyagüe, J., Salido, F., Salinas, M., Manteca, M. and García, D. (2019). [online] Koodous.com. Available at: https://koodous.com/ [Accessed 16 Mar. 2019].

Argus Cyber Security Lab (2019). [online] Android Malware Dataset. Available at: http://amd.arguslab.org/ [Accessed 16 Mar. 2019].

Pomilia, M. (2016). A study on obfuscation techniques for Android malware. Master. Sapienza University of Rome.

Grace, M., Zhou, Y., Zhang, Q., Zou, S. and Jiang, X. (2012). RiskRanker: scalable and accurate zero-day android malware detection. In: MobiSys '12 Proceedings of the 10th international conference on Mobile systems, applications, and services. [online] pp.281-294. Available at: https://yajin.org/papers/mobisys12.pdf [Accessed 1 Nov. 2018].

Chekina, L., Mimran, D., Rokach, L., Elovici, Y. and Shapira, B. (2012). Detection of Deviations in Mobile Applications Network Behavior. In: Annual Computer Security Applications Conference. arXiv.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. Network and Distributed System Security Symposium. (2014).

Zou, S., Zhang, J. and Lin, X. (2014). An effective behavior-based Android malware detection system. Security and communication networks, [online] pp.2079-2089. Available at: https://onlinelibrary.wiley.com/doi/full/10.1002/sec.1155 [Accessed 1 Feb. 2019].

Sun, L., Li, Z., Yan, Q., Srisa-an, W. and Pan, Y. (2016). SigPID: significant permission identification for android malware detection. In: 2016 11th International Conference on Malicious and Unwanted Software (MALWARE).

Mendoza, M. and Zhu, M. (2017). Paranoid Android : Android Malware Classification Using Supervised Learning on Call Graphs. semantic scholar.

Xu, K., Li, Y., Deng, R. and Chen, K. (2018). DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks. In: European Symposium on Security and Privacy (EuroS&P). IEEE.

Li, C., Zhu, R., Niu, D., Mills, K., Zhang, H. and Kinawi, H. (2018). Android Malware Detection based on Factorization Machine. [ebook] arXiv. Available at: https://arxiv.org/abs/1805. [Accessed 1 Feb. 2019].

Salem, A. (2018). Stimulation and Detection of Android Repackaged Malware with Active Learning. [ebook] arXiv. Available at: https://arxiv.org/abs/1808.01186 [Accessed 1 Feb. 2019].

Wang, S., Chen, Z., Yan, Q., Yang, B., Peng, L. and Jia, Z. (2019). A mobile malware detection method using behavior features in network traffic. Journal of Network and Computer Applications, 133, pp.15-25.

Tian, K., Yao, D., Ryder, B., Tan, G. and Peng, G. (2017). Detection of Repackaged Android Malware with Code-Heterogeneity Features. IEEE Transactions on Dependable and Secure Computing. [online] Available at: https://ieeexplore.ieee.org/document/8018581 [Accessed 8 Feb. 2019].

Av-comparatives.org. (2019). Android Test 2019 – 250 Apps. [online] Available at: https://www.av-comparatives.org/tests/android-test-2019-250-apps/ [Accessed 21 Mar. 2019].

Threatfabric.com. (2018). MysteryBot; a new Android banking Trojan ready for Android 7 and 8. [online] Available at: https://www.threatfabric.com/blogs/mysterybot__a_new_android_banking_trojan_ready_for_android_7_and_8.html [Accessed 15 Jan. 2019].

Fedler, R., Schutte, J. and Kulicke, M. (2013). AN EVALUATION OF ANDROID ANTIVIRUS APPS. On the Effectiveness of Malware Protection on Android. [online] Fraunhofer. Available at: http://www.linkality.org/res/uploads/042013-Technical-Report-Android-Virus-Test.pdf [Accessed 1 Mar. 2019].

Lueg, C. (2017). 8,400 new Android malware samples every day. [online] gdatasoftware. Available at: https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day [Accessed 1 Feb. 2019].

Kabakus, A. and Dogru, I. (2018). An in-depth analysis of Android malware using hybrid techniques. [ebook] Digital Investigation, pp.Volume 22, Pages 25-33. Available at: https://www.sciencedirect.com/science/article/pii/S1742287617303183 [Accessed 4 Nov. 2018].

Arzt, S. (2018). FlowDroid. GitHub.

rednaga (2016). APKiD. GitHub.

pxb1988 (2015). dex2jar-2.0. GitHub.

Wiśniewski, R. and Tumbleson, C. (2015). APKtool. GitHub.

JD-GUI. (2015). Java.

pjlantz (2014). DroidBox. GitHub.

Abraham, A. (2018). MobSF. Github.