Black Box Analysis of Android Malware Detectors

Guruswamy Nellaivadivelu
San Jose State University

Black Box Analysis of Android Malware Detectors

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Guruswamy Nellaivadivelu

May 2017

The Designated Project Committee Approves the Project Titled

Black Box Analysis of Android Malware Detectors

by

Guruswamy Nellaivadivelu

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

ABSTRACT

Black Box Analysis of Android Malware Detectors

by Guruswamy Nellaivadivelu

Code obfuscation can make it challenging to detect malware in Android devices. Malware writers obfuscate the code of their programs by employing various techniques that attempt to hide the true purpose of the program. Malware detectors can use a number of features to classify a program as a malware. If the malware detector uses a feature that is obfuscated, then the malware detector will likely fail to classify the malware as malicious software. In this research, we obfuscate selected features of known malware and determine whether the malware can still be detected by a given detector. Using this approach, we show that we can effectively perform black box analysis of various malware detectors.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to Dr. Stamp for his valuable guidance and supervision throughout the project work. I am also extremely thankful to him for teaching me interesting new concepts and techniques in the field of security during the entire course of my graduate program. I am also grateful to my committee members for reviewing my work and providing constructive feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

The volume of Android malware is increasing exponentially. Indeed, in the second quarter of 2016, 3.5 million samples of Android malware were detected [1]. This rapid increase in Android malware has placed the focus on Android security and made it imperative to develop more efficient defensive tools for combating such malware. One of the challenges faced in this area is the use of code obfuscation techniques. Code obfuscation is a method of altering code to hide its actual purpose, without significantly altering its performance. There are many ways of obfuscating source code in an Android environment. Several software applications that are available off the shelf can be used to achieve different levels of code obfuscation [2]. In order to address the problem of strengthening malware detector's strength, there are two fundamental questions that need to be addressed, as highlighted by Christodorescu et al. [3]. The first question is to gauge the resilience of a malware detector against code obfuscation. This will also help us in understanding the strength of the malware detetctor in detecting variants of known malware families. The other question is the possibility of identifying the detection algorithm used by a malware detector. By studying the behavior of malware detectors and how they respond to different obfuscation techniques, a malware writer can uncover ways to beat the antivirus program. Ultimately, we want to gauge how well a malware detector will perform against obfuscated code.

Code obfuscation is the process by which source code is manipulated to hide its true intentions. Code obfuscation is increasingly becoming a common tool to avoid detection by traditional malware detectors. There are many different types of code obfuscation. The most basic type of code obfuscation involves the encryption of all the strings that are used in the code. This overrides the detection mechanism of most of the traditional malware detectors. Some advanced malware detectors account for this encryption and are able to identify malware files. There are a host of other obfuscation techniques that can be employed by malware writers. Some of these include the obfuscation of function calls, permission hiding, and insertion of dead code.

The challenges associated with code obfuscation primarily deal with the problem of maintaining the core functionality of the code, while making it difficult for malware detectors to detect their true purpose. This challenge becomes easier for malware writers when dealing with Android malware. The reason for this is associated with the permission levels of applications running on Android platform. Unlike anti-virus programs that run on computers, the Android system provides the same set of permission levels to the anti-virus application and the application that is being scanned. This is a major limitation for malware detector writers. With the advent of sophisticated encryption techniques, it has become very difficult to different between benign and malicious applications of obfuscation techniques. The primary objective of this project is to make malware detectors more responsive to the code obfuscation techniques employed by malware writers. By doing so, we can attempt to identify the malware features that are used by a malware detector in its classification algorithm. We can also try and modify an existing malware detector to see if we can overcome the limitations. In order to achieve this, we propose a theoretical approach. In this approach, we attempt to isolate the features that contribute to malware detection. Once, we have this information, we can attempt to modify an existing malware detector to overcome these limitations. The malware detector should employ "de-obfuscation" techniques before analyzing any malware. An intelligent malware detector should be able to sense the type of encryption or obfuscation technique being employed and use the corresponding "de-obfuscator" to nullify the effects of the obfuscator.The first step in this implementation will be the identification of the factors in a malware that are taken into consideration by a malware detector. To achieve this, we will begin by encrypting various parameters of a malware and running it through a malware detector [4]. By following this approach, we can identify the exact scenario when a malware is no longer classified as a malware by our malware detector. Once we identify the features that are required by a malware detector, we will use this information to make the malware detectors process the obfuscated part of the code as well. This will make our malware detector more robust and improve their performance.

In Chapter 2, we look at the previous work that is done with regards to malware detection in Android. We explore the various detection mechanisms and approaches that has been discussed so far. After looking at the background work, we delve into code obfuscation in Chapter 3 and understand the basic terminologies associated with code obfuscation. We also look at the impact of obfuscation in general, and then more specifically, their impact on malware detectors. After understanding the basics of code obfuscation and malware detectors, we move on to the current threats and defenses in the Android operating system in Chapter 4. In Chapter 4, we glance at the growing dominance of the Android operating system in the mobile phone space and the importance of this particular operating system in our lives. The motivation behind selecting the Android OS for this project is understandable from Chapter 4. The obfuscators to be used in this project, and their functionalities are explained in Chapter 5. Chapter 6 clearly lists the software requirements for this project and also talks about the necessary technologies for setting up the experiment. The results of the experiment are summarized in Chapter 7. The factors that contribute to the conclusion being drawn from this experiment are detailed in Chapter 6. We finally consolidate the results and discuss the future course of the project work in Chapter 8.

CHAPTER 2

Previous Work

In this chapter, we present the results of a literature survey that was performed to identify the current state of obfuscation mechanisms and their impact to the field of code obfuscation. We find that code obfuscation has been an area of interest in the field of cryptography and traditionally, obfuscation techniques have been used to achieve reverse engineering protection. On the other hand, a lot of malware have obfuscated code to avoid detection by anti-virus programs.

2.0.1 Code Obfuscation and Malware Detectors

The efficiency of malware detectors against code obfuscation has been a point of discussion amongst malware researchers for a very long time. A lot of research has been done on the robustness of malware detectors against high levels of obfuscation. The issue of malware detector's strengths against obfuscated malware had been discussed as early as 1996, as can be seen in the quote by S. Gordon and R. Ford [4]:

"The evaluation of anti-virus software is not adequately covered by any existing criteria based on formal methods. The process, therefore, has been carried out by various personnel using a variety of tools and methods."

2.0.2 Program Obfuscation

There has been a lot of theoretical research on the different aspects of obfuscation and on ways to improve it. Most of this research has been successful in arriving at a conclusion on the efficiency of the cryptographic problems of encryption, authentication and protocol [5]. But the problem of program obfuscation has remained an area within cryptography in which theoretical research has been inadequate. In their seminal paper on program obfuscation, Barak et al. [5] propose to represent program obfuscation as below: An obfuscator O is said to be an efficient compiler if it takes as input a program P and produces a program $O(P)$ and satisfies the following two conditions:

1. Functionality: $O(P)$ computes the same function as P

2. 'Virtual Black Box'property: Anything that can be efficiently computed from $O(P)$ can also be computed by P.

The paper by Christodorescu et al. [3] lists various ways to test and achieve program obfuscation in general. A detailed analysis of the various obfuscation methods is also discussed in the paper. One interesting angle explored by the paper deals with assigning mathematical equations to measure the effectiveness of the individual obfuscators. This lets us quantify the different obfuscators and rank them against each other. One of the evasion methods employed in malware obfuscation is polymorphism. It is a method by which a program evades various detection tools by mutating into different forms. In the paper by Rastogi et al. [6], the authors develop and propose a framework called 'DroidChameleon' that provides a way to transform Android applications into different forms with minimal user involvement. As shown in Figure 1, the authors apply various transformations on a malware sample dataset. The output of all these transformations are processed by a malware detector (referred here as Anti-malware). The input to the anti-malware is processed sequentially. After each

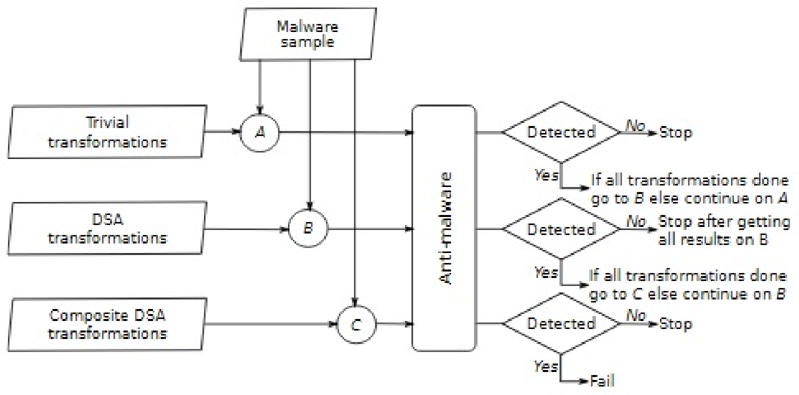

Malware sample
Trivial transformations
DSA transformations
Composite DSA transformations
A
B
C
Anti-malware
Detected
No
Stop
Yes
If all transformations done go to B else continue on A
Detected
No
Stop after getting all results on B
Yes
If all transformations done go to C else continue on B
Detected
No
Stop
Yes
Fail

Figure 1: Evaluating anti-malware

transformation, the anti-malware‘s output is evaluated and if the malware detection fails, the next level of transformation is applied. This helps rank the various malware detectors against each other for accurate analysis.

2.1 Obfuscation in Android Malware

A report by Google stated that a majority of malware detectors work as a binary classifier [7]. They classify an application as a malware or a benign file. In order to effectively eliminate malicious applications, it is important that malware detectors do more than just identify malware. They should be able to isolate the core parts of the application that perform the malicious acts and work at fixing the loopholes that let the program act in a malicious way. More recent malware applications employ a variety of tricks, in addition to traditional code obfuscation mechanisms. For instance, a variant of Android malware, known as Android/BadAccent, is a known banking Trojan, that steals credentials used in banking applications [8]. A variant of this malware used a mechanism known as ‘Tapjacking’ to extract the credentials from the users. In this form of attack, a screen is displayed to the user, while a second screen is hidden behind the actual visible display [9]. When a user clicks a button on the screen, assuming it to be the one that is displayed, the underlying screen gathers the input and processes the command. This is a common method of gathering details from unsuspecting users.

2.1.1 Statistical Anaylsis Techniques and Android Malware

One widely used approach for analyzing malware samples is the usage of statistical methods. In such methods, the Android executable file (with the extension apk), is decompiled to get the original source code. Due to the Android operating system being written in Java, it is easy to reverse engineer an apk file to retrieve the source code. This opens up many opportunities for performing statistical analysis on the obtained raw data. This also lets a researcher perform various operations on the source code, and then repackage it back into an apk. In the approach known as AndroSimilar, Faruki et al. [10] propose a new algorithm known as AndroSimilar, that takes into consideration various features that are known to be present in malware alone. The AndroSimiar approach [10], as shown in Figure 2, decompiles an apk file and repackages it after feature extraction. To extract the features, the algorithm incorporates apps from the Google Playstore and other third party applications. These features are normalized and fed into a signature generation engine, that provides a unique signature for each malware. This is used as reference for detecting future malware applications.

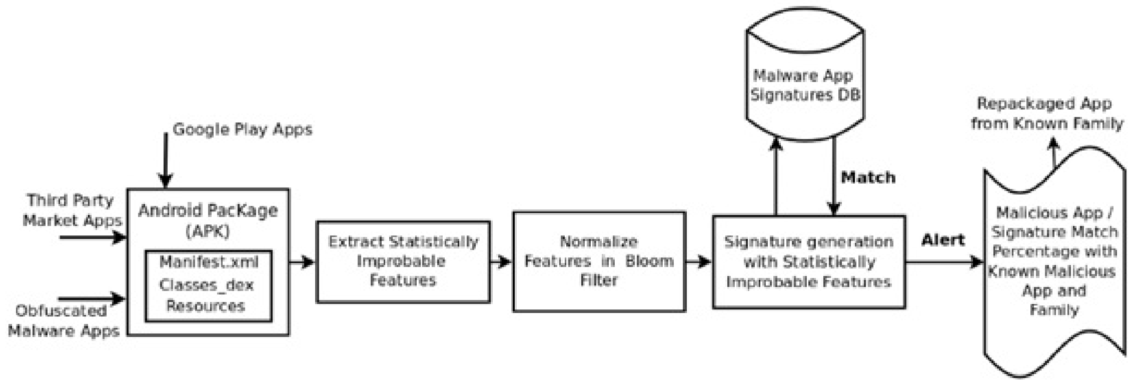

Google Play Apps
Third Party Market Apps
Obfuscated Malware Apps
Android PacKage (APK)
Manifest.xml
Classes_dex
Resources
Extract Statistically Improbable Features
Normalize Features in Bloom Filter
Signature generation with Statistically Improbable Features
Malware App Signatures DB
Match
Alert
Repackaged App from Known Family
Malicious App / Signature Match Percentage with Known Malicious App and Family

Figure 2: AndroSimilar

2.2 Conclusion

Malware in mobile devices is no longer a problem confined to labs and research areas. The rapid increase in access to computers has helped malware writers create specific, targeted programs that perform with high efficiency and exploit vulnerabilities in different operating systems. The amount of research being done in malware analysis and, more specifically, in Android malware, is in the right direction. In the fight against sophisticated metamorphic malware, it is imperative that the malware detector is better than the malware creator. In this paper, we have explored various work, that dealt with the different aspects of malware obfuscation and ways to overcome the shortcomings in today's version of malware detectors. The future of malware looks very bright and it is hoped that the malware detectors of the future will be up to the task at hand.

CHAPTER 3

Code Obfuscation

Code Obfuscation is a technique by which programmers have deliberately sought to make the functionality of their code less obvious. This technique has been used by programmers to achieve various additional objectives. Code obfuscation can be used to achieve a myraid of objectives. These include prevention of reverse engineering, protection of intellectual property, and reducing the size of an executable. In some benign scenarios, an executable is obfuscated to protect the various licensing mechanisms used in them. Obfuscators are also a good way to restrict unauthorized access to files by people who might try to use dubious tools to incorporate malicious code into files.

We will look at the history of code obfuscation to appreciate the relevance of code obfuscation in today's software development perspective. With growing interest in various obfuscation techniques, and the ease of availability of obfuscators dedicated to different operating systems, this would help us in understanding the rapid growth in this area and appreciate the urgent need for various countermeasures against this approach.

3.1 Growth of Obfuscation in Software Development

Code obfuscation has been historically associated with malware development, than with benign software development. Some of the earliest examples of attempts at obfuscation in malware can be found in the "Brain Virus" . In this variant of the malware, the malicious program would display unaffected disk partitions to users attempting to access partitions that the virus had corrupted. Although the code in itself was not encrypted, the behavior of the virus shows attempts at hiding its true usage.

In the same year, the Cascade virus was released to the world. This was an early variant of malware to use encryption to hide its true purpose. The earliest strains of obfuscated malware used a simple encryption-decryption routine to perform the decryption tasks. As the malware detectors of the time were not sophisticated enough to detect the encrypted part of the code, this simple obfuscation technique enabled a lot of malware programs to slip away undetected. This is a serious disadvantage in the design and implementation of malware detectors. We would be exploring more such flaws with the implementation of malware detectors in this project.

With the advent of advanced malware detectors and improvement in statistical analysis techniques, the level of obfuscation in malware increased. Polymorphic malware uses a very high level of encryption technique to obfuscate its contents. A polymorphic malware changes the encryption in itself and provides very few traces of a signature. If a malware is truly polymorphic, then there will be no consistency between any two iterations of the same program and it would be virtually impossible to detect them using traditional signature matching techniques.

3.2 Malware Detectors

Malware detectors came into existence with the advent of different malicious programs. Before the rapid growth of the internet, malware detectors were only capable of performing scans based on signatures of known virus programs. This static analysis technique meant that new virus would be out in the wild for some time before the malware definitions of the individual anti virus programs could be updated. With the introduction of the world wide web, the antivirus industry expanded into dynamic analysis and cloud based malware detectors. Firewalls, online scanning, and virtual machines started being increasingly used to identify malware. One major shortfall of anti virus programs is their inability to detect polymorphic virus. In general, many antivirus programs employ signature detection for identifying malware. In addition to this most common approach, heuristics based detection and rootkit based detection are also employed to detect virus programs. Along with these approaches, active scanning approaches like on-access scanning is also used to detect programs that might attempt unauthorized operations. We discuss these methods and detection mechanisms in detail in this chapter.

3.2.1 Signature Based Detection

This is one of the most basic methods of malware detection that is still in use today. When a new strain of malware is detected in the "wild", antivirus firms analyze it and extract a "signature" from it. This signature extraction can either be done manually or by using automated signature detection techniques [11]. Once a signature is detected, it is updated into various malware definitions of antivirus software. Although this method is effective against generic malware, it is highly ineffective against oligomorphic, polymorphic and metamorphic malware. These are variants of malware that encrypt itself with each iteration. In this project, we attempt to identify the various factors that contribute to malware detection and their importance in overcoming the signature detection method.

3.2.2 Heuristics Based Detection

In Heuristics based detection techniques, a single signature or pattern is used to detect multiple malware belonging to the same family. Such techniques rely on the fact that multiple malware are created from a single malware. Thus, successfully creating a signature for a base family will result in the detection of all malware related to that particular family.

3.2.3 Rootkit Detection

A rootkit is a type of software that attempts to gain administrator privileges in a system without the knowledge of the user running it. In many cases, the rootkits contain software within them that becomes undetectable to antivirus programs. Rootkits usually have full administrative access and also have the ability to hide themselves from the list of running processes. Modern antivirus software scans for rootkits in specific, to detect them. It is very difficult to remove a rootkit when compared to other generic malware programs.

3.2.4 On-Access Scanning

In this method, the antivirus program looks out for any threats that might happen on a real-time basis. The antivirus monitors the system in which it is installed and looks for suspicious activity whenever the computer's memory is loaded with fresh data from the storage disks. This might happen when a USB drive is inserted, an email attachment is opened or a even when an already existing file is opened by a user or a program. This type of scanning is more effective as it does not rely solely on malware definitions to detect viruses.

CHAPTER 4

Threats and Defenses in the Android Operating System

Before we dwell deep into code obfuscation in Android, we look at the various malware detectors for the Android operating system. We also look at the rapid proliferation of the Android OS and the reason for selecting Android as the focus of study in this experiment.

4.1 Android Malware Detectors

With the rise of the Android Operating systems, the amount of malware associated with it has also risen significantly. From a market share of 2.8 % in 2009 [12] , Android captured about 75% of the market in 2012 [12]. As shown in figure 3 , we can see that the growth and adoption of Android has been very steep. This rapid proliferation of Android resulted in an equally rapid rise of Android malware.

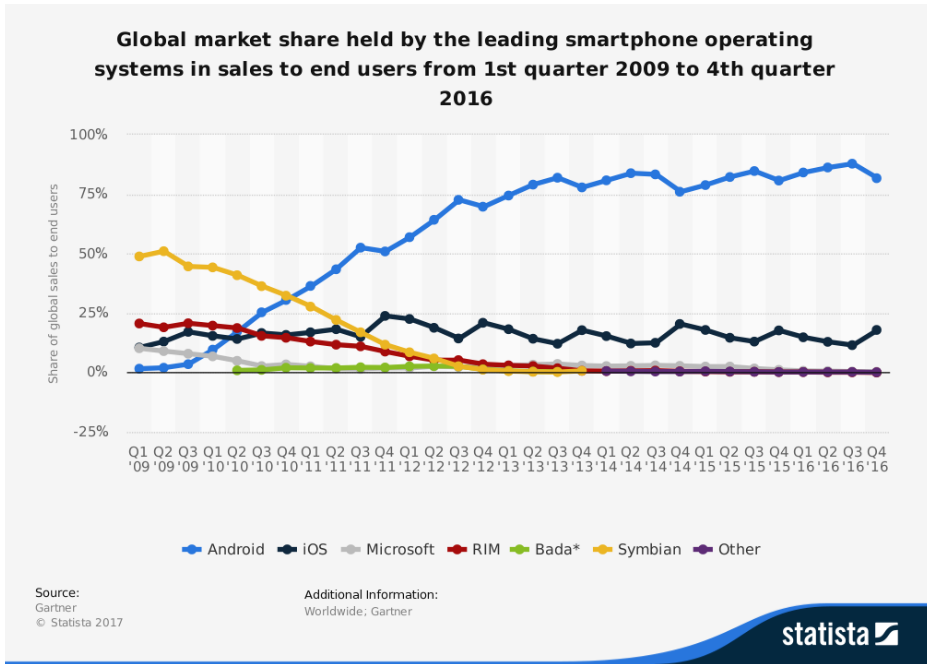
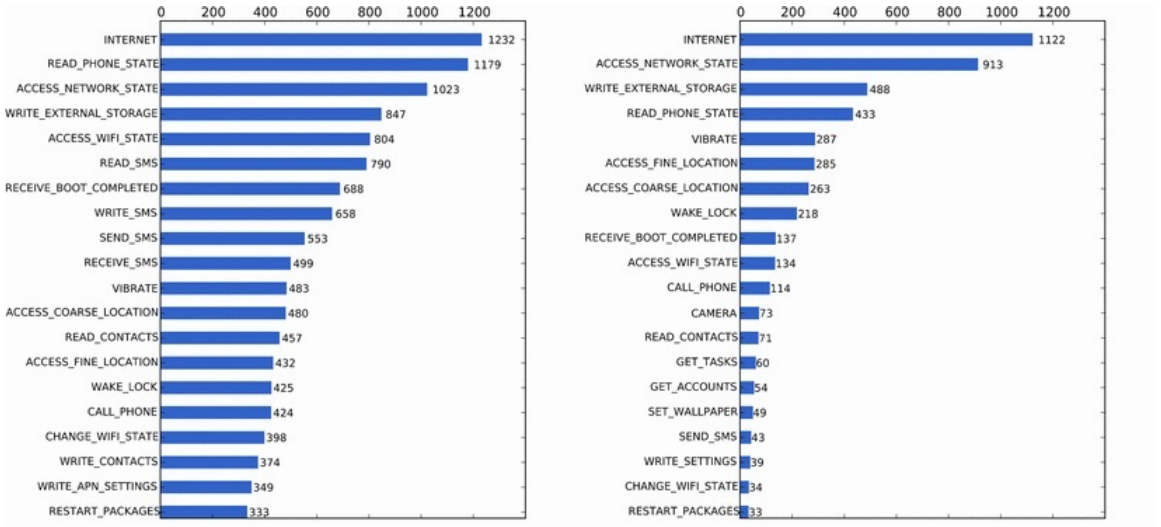


Figure 3: Market Share of mobile operating systems



(a) Top 20 Permissions Requested By 1260 Malware Samples

(b) Top 20 Permissions Requested by 1260 Top Free (Benign) Apps on the Offical Android Market

Figure 4: Top 20 permissions in Android in 2012

With the increase in the number of Android malware being released to the wild, their level of sophistication also increased. Android malware detectors used the number of permissions requested by an app to determine its legitimacy. In the schematic represented in Figure 4, Zhou et al. [13] support the fact that both, benign and malicious applications, have very similar permission requests. Due to this, using access requests as a measure for classifying Android applications became ineffective. All the malware programs plaguing the Android operating system can be classified into four categories based on the basis of their primary activity [13]. Privilege Escalation, Remote Control, Monetary Loss, and Information Collection are the various sub categories under which any Android malware can be classified.

4.1.1 Privilege Escalation

In this type of attack, the malicious app that is installed on a device, attempts to grant itself additional privileges than the one it requires. This is achieved by using known exploits in the Android operating system.

4.1.2 Remote Control

A very high percentage of malware attempts to use the compromised device as a remote bot. In some malware families, the remote URL that is being used to control the device is encrypted. Such encryption makes it very difficult to detect these types of malware and this will be a primary area of focus in this thesis.

4.1.3 Monetary Loss

A very direct way of monetizing malware is to make unsuspecting users subscribe to services that cost a lot of money. Such services are run by the malware perpetrators and will enable them to charge the infected devices' owners money for services that they are not aware of. To achieve this, some malware use the remote control to push down numbers of services to the devices and then enroll them.

4.1.4 Information Collection

Many malware programs attempt to collect the personal information of users. Such personally identifiable information makes it easy for scamsters to dupe people using various other schemes. Malware belonging to this family tries to steal personal information of the compromised device's owner, as well as the details of people in their contact lists. This information is then sold through different means to interested parties.

4.2 Android Malware Detection Limitations

One of the major limitation of malware detection in Android is the limited processing power of the devices running Android. Due to processing and memory constraints, generic malware detection has to be restricted to static analysis techniques. In general, all the existing Android security solutions can be classified into *Static Analysis* and *Dynamic Analysis* [14].

4.2.1 Static Analysis

Static Analysis is a technique in which the an application is evaluated for its trustworthiness by disassembling and checking its source code. The application is not executed for this analysis. Once an application is marked for scanning, various statistical analyzing approaches are used to classify the file. Some of the most commonly used static detection methods are discussed in the next few subsections.

4.2.1.1 Signature Based Detection

Signature based detection is a type of static analysis technique. In this method, a virus is examined by extracting its signature and then comparing it with signatures from known malware. The limitation of this technique is that it is incapable of detecting unknown malware types. The signatures of known malware are stored in a signature database. In addition to this, the signature database also requires that it is updated constantly. Without an up-to-date signature database, most of the prevalent malware could slip through undetected. This is difficult in the case of Android Malware detectors as the device possess limited memory and it would be infeasible to store all virus definitions on the device. If the virus definitions were to be moved to a remote server, it would use up considerable amount of data traffic for performing the validation. These are some serious limitations that hinder traditional signature matching techniques.

4.2.1.2 Permission Based Detection

This is a straightforward approach to detecting malware in Android systems. In this method, the number of permissions an application requires is used to determine its classification as a malicious or a benign file. Some research has been done in this area wherein the Android Manifest file is analyzed for extracting information [15] about the permissions requested by the application. This information is used to assign a score of relevancy to the permissions requested. This score is then compared against a threshold for determining the malicious intent of an app. There are variations to this technique and some methods yield better results than the others. This method is a very quick way of determining the malicious nature of applications. But a serious limitation of this method is that it does not analyze the source code or the working of the app. Only the Manifest file is analyzed. A lot of malware apps use permissions similar to the benign apps. Hence, permissions based detection should be used in conjuction with a second confirmation method to validate an app.

4.2.2 Dynamic Analysis

In this method, the application is executed and it is analyzed during the runtime. It becomes very easy to identify sections of code or execution blocks that were missed during the static analysis of an application. Dynamic analysis methods are also effective against obfuscation and encryption techniques.

4.2.2.1 Anomaly Based Detection

An application is executed and the system calls generated by it are recorded in a log. This log is then sent for analysis to a remote server, where the various behavior of malware are recorded. Using that as a basis, the log files are analyzed, and the results are aggregated. This result, in collaboration with other techniques are used to classify the file as malicious or not.

4.2.2.2 Emulation Technique

Yan et al. [16] propose a technique in which a virtual machine is used to analyze an application. In common virtual machine based detection techniques, the anti-malware program and the malware execute in the same environment. This makes them detectable to each other. In the platform presented by Yan et al. [16], the antimalware, *DroidScope*, stays out of the execution environment and monitors the execution as a whole. This enables it to detect the malware without being detected by the malware.

CHAPTER 5
Android Obfuscators

In this chapter, we use different obfuscators to modify parts of an android malware. By systematically obfuscating different parts of the code, we can gain insight into the parts which contribute most to the detection of malware. Once we have this information, we can then determine efficient ways to make the malware detectors more robust and be less resilient to code obfuscators.

5.1 Experiment

For this project, we use a tool called AAMO (Another Android Malware Obfuscator) [17]. This tool gives us various obfuscators for use with our experimentations. The obfuscators can be used independently or in combination with other obfuscators to increase their effectiveness. Using this tool, we decompile a android file, perform obfuscation operations on them, and recompile the file again. In this experiment, we use the source code provided by the developers of AAMO [17] and available at [18]. This tool forms the basis of the work presented in this thesis. The steps involved in this are detailed below:

1. Obtain an APK file.
2. Decompile the APK file into Smali.
3. Get the list of obfuscators passed into the program.
4. Apply the obfuscators one after the other on the decompiled apk file.
5. Repackage the decompiled file into an APK.
6. Sign the APK file to maintain its integrity.

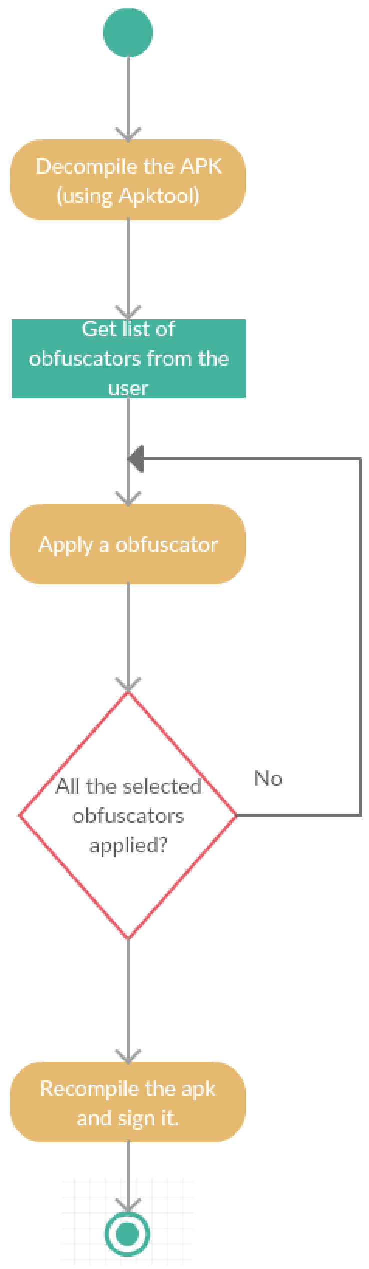

Decompile the APK (using Apktool)
Get list of obfuscators from the user
Apply a obfuscator
All the selected obfuscators applied?
No
Recompile the apk and sign it.

Figure 5: Experiment flow

Performing the above steps ensures that the apk file is not corrupted and its usage is not affected. We perform this to make it difficult for a malware detector to detect the apk file as a malicious one. The entire flow of the experiment is depicted in Figure 5.

As shown in Figure 5, the final encryption would let the malicious file be signed with a valid signature and thus eliminating any traces of the apk file having been compromised.

5.1.1 Uses of the obfuscator

Using the obfuscator in this step has various advantages for our experiment. One of the primary uses is to make the job of the malware detector more difficult. Since most of the malware detectors do not take into account polymorphic and oligomorphic malware, using obfuscators will let us know which parts of a malware factor into the detection score computed by individual detectors. In this experiment, we use 14 obfuscators to test out the resilience of the malware detectors as listed in Table 1.

Table 1: Android Obfuscators

Count	*Obfuscator Name*
1	Resigned
2	Alignment
3	Rebuild
4	Fields
5	Debug
6	Indirections
7	Renaming
8	Reordering
9	Goto
10	Arithmetic Branch
11	Nop
12	Lib
13	Manifest
14	Reflection

These obfuscators enable us to test the various aspects of a apk file and help us determine the ones that are really useful to a malware detector. When a particular obfuscator is run, it runs a function that is specific to that particular obfuscator and applies that function to all the parameters that match the criteria for that specific obfuscator. Each of the obfuscator is discussed here in detail.

5.1.1.1 Resigned

This obfuscator decompiles an apk and just resigns the apk file after compilation. Not much change is done to the application file in itself. The purpose of this obfuscator is to attempt defeating malware detectors that try to use signatures of certain known malware sources to classify a malicious file.

5.1.2 Alignment

This obfuscator makes use of the zipalign utility of android. Zipalign is a tool that is used to provide optimization techniques to APK files. The tool causes all uncompressed data within the APK to start with a particular alignment relative to the file's beginning. The Alignment obfuscator changes this alignment before recompiling the apk file.

5.1.3 Rebuild

This obfuscator rebuilds the application file without performing any changes. The unpacking and repackaging of the apk file affects the timestamp, signature of the apk and other factors that help in identifying the origin of the file. Some smart malware detectors are able to detect these changes and do not let the file pass through it.

5.1.4 Fields

This is a relatively simple obfuscator that just renames the fields that are used in the application. This is done after the decompilation of the apk file. The smali is analyzed for locating the fields that are used in the source code and these are renamed.

5.1.5 Debug

The debug obfuscator removes all information related to debug from the files. This is performed not only on the smali file, but throughout the source code as well.

Without the debug information, the APK file becomes slightly different from the original file. Removal of the debug information also alters the size of the file and makes it different.

5.1.6 Indirections

Call indirections is an advanced obfuscation method in which various function calls are directed through different values. The obfuscator performs operations such as changing the register count, changing a method call and also redirecting all calls to the methods. This obfuscation completely changes the control flow of an application and makes it difficult to detect using a comparison model in dynamic analysis as well.

5.1.7 Renaming

All the variables in the sourcecode are renamed to different values. This is exactly like using substitutions to hide the original values. Renaming is also advantageous when certain signature and pattern matches are based on the names of the variables and functions.

5.1.8 Reordering

Using reordering will let us change the order of the code in the application. The obfuscator changes the location of certain parts of the code and adjusts the calls to it accordingly. This makes it possible to evade signature based detection methods if the signature is based on the order of instructions or if it is based on the DEX opcodes.

5.1.9 Goto

In order to modify the control-flow structure of the application, forward and backward jumps are inserted into the code. These unconditional jump statements will be executed irrespective of how the program is run. This widely alters the flow and will make it very difficult to detect using conventional methods.

5.1.10 Arithmetic Branch

A constant value, known to the obfuscator, is used to achieve this obfuscation. This constant value is not known to the compiler. Using this constant value, the obfuscator is able to control the flow of execution of the program. The compiler assumes that either of the branches could be possible as the value for deciding the flow of control is not known. This is applied to methods with more than 2 parameters.

5.1.11 Nop

This is a classical and an easy way to obfuscate a program. In this, a "no-operation instruction" (known as a "NOP") is inserted into the source code. The number of such instructions inserted is randomized. These are inserted into methods to make them bloated and delay the execution time.

5.1.12 Lib

MD5 hashing is used to rename the file and path names. A proxy method is created and used to handle the decryption of the values, when it is required by the system.

5.1.13 Manifest

The AndroidManifest.xml file is modified by this obfuscator. The manifest file contains important information related to the application's usage and permissions. This obfuscator opens up the file and encrypts the values for the resources and also replaces the characters in user defined identifiers.

5.1.14 Reflection

This obfuscator acts similar to the code reodering obfuscator. The reflection obfuscator takes advantage of the Android dynamic code loading API. All the static method calls are converted into reflection calls and the the reflect method is invoked on a string that contains the target method's name.

CHAPTER 6

Experiment

6.1 Environment Setup

Due to the various different types of software used in the experiment, it is important to have the correct version of each software installed. As shown in Figure 5, each APK will have to be decompiled into its source code, before going through the obfuscation process. To achieve this, we use a program called apktool[19]. A list of various software and their versions are listed in Table 2.

Table 2: List of software required and their versions.

Number	Software	Version
1	Java	1.8.0_45
2	Python	2.7.11
3	Apktool	2.2.1

The given applications are interdependent on each other for this experiment. The AAMO framework is written in Python and uses various Python libraries to execute. The decompilation of the APK files is achieved using the Apktool. Apktool requires a java virtual machine to execute. It is imperative that this version of Apktool be maintained for repeating the experiments presented in this work as the source code of AAMO has been modified to fit this version of the tool.

6.2 Dataset and Malware Detectors

In order to successfully evaluate and analyze the malware detectors various experiments were performed using known Android malware. Once the android malware were finalized, the obfuscators were chosen to increase the difficulty of malware detection. A sampling was performed with a handful of malware. Using this sampling, the obfuscators to be applied were selected and then applied to a wider dataset. The

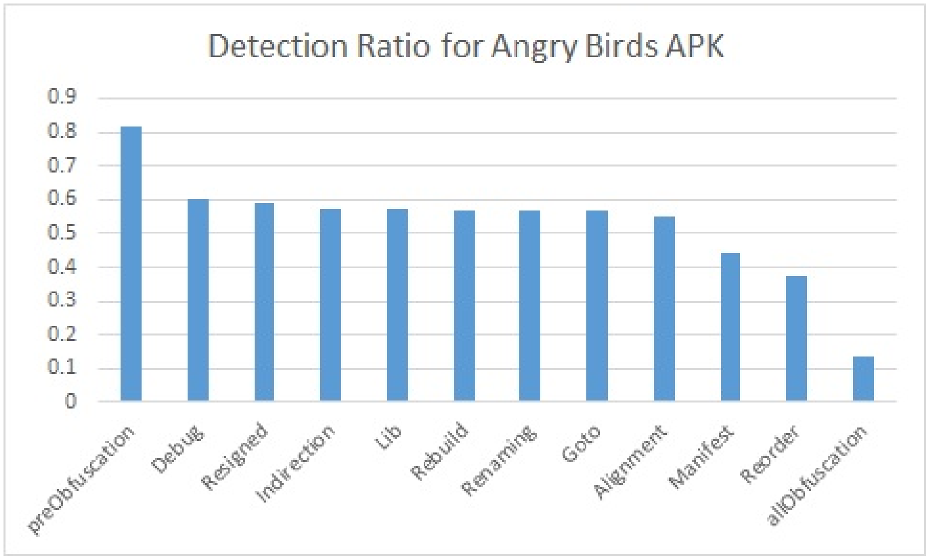


Figure 6: Sampling Results

results of this sampling are shown in Figure 6. In Figure 6, a sample file, "Angry Birds", was used to test the effect of the various obfuscators. This file, a malicious version of the popular game, is a Trojan variant that steals the contact information, and has the ability to send text messages without the user's permission. Before applying any obfuscator, the file had a detection ratio of 0.819. When we apply the obfuscators, the detection ratio drops steadily. In Figure 6, the *x-axis* represents the different obfuscators that were used. We can see that the detection ratio almost remains constant for all the obfuscators, except for the manifest and reorder obfuscators. The functioning of these obfuscators are defined in chapter 5. This hints at the fact that many malware detectors just perform an analysis on the AndroidManifest.xml file to classify the file as a malware. Due to this, when the manifest obfuscator is applied, the detection ratio drops. When we apply all the obfuscators on the file, the detection ratio drops significantly. This is shown in the Figure 6's 'allObfuscation' bar.

It can be seen from the results that only some obfuscators contribute effectively to hindering the detection ratio of malware obfuscators.

6.3 Dataset

The Contagio dataset was used to perform the various experiments in this project [20]. All the samples used for experimentation are malicious files. The files were classified as malicious by various means and the contagio data dump also certifies the files as being malware.

6.3.1 Malware Files Selection

Known malicious files were used for performing the experiments in this project. The reason for using malware for the experiments was to understand how each obfuscator would help the malware in evading detection by malware detector. All the test samples were caught by at least one of the malware detectors and many of the samples were incorrectly classified as benign files, once the obfuscation was complete.

6.3.2 Other Datasets

Previously, experiments have been performed on malicious files belonging to other datasets. Before we delve into the results of the experiment performed in this work, we look at how obfuscators affect the detection ratio of various malware detectors.

6.4 Malware Detectors against Code Obfuscation

A single malware detector is unlikely to give us a substantial result. This is because various malware detectors use different techniques for analyzing malware. If a single malware detector were to be used as a benchmark, then we would either get excellent detection scores or the malware detector would fail in a very poor way. To overcome this shortcoming, a single obfuscated file is scanned by several malware detectors simultaneously. Instead of manually uploading the files to different malware detectors, we make use of VirusTotal [21] and other similar virus scanning providers.

CHAPTER 7

Results of Experiments

The various obfuscators defined earlier were iteratively applied to malware samples from the Contagio dataset. Based on the results obtained from VirusTotal, the obfuscators were selected for further application.

7.1 Observations

The VirusTotal website uploads a malware file to its database and then performs a scan using the various malware detectors associated with the website. Each uploaded file is hashed and stored in the database to reduce duplicate efforts and minimize scan times. Due to this behavior, each time a file is loaded into the website to be scanned, the website will prompt if a similar file was scanned earlier. It was observed that as the number of obfuscators employed increased, the similarity between the obfuscated and un-obfuscated applications decreased. If more than 2 certain obfuscators were applied, the VirusTotal website would not recognize the file as a previously recognized file. This observation was consistent throughout the different experiments conducted.

7.2 Steps for Analyzing Malware Detectors

The experiment was performed with certain operations being repeated in an iterative manner. The obfuscated malware files were prepared in advance. The steps are as follows:

- Scan a malicious file using VirusTotal.
- Record the detection ratio.
- Apply obfuscator(s) on the selected malware file.
- Scan the obfuscated file using VirusTotal again.
- Record the new detection ratio.

Repeating the above steps helped us detect how robust and efficient malware detectors are. Ideally, the malware detector should not be affected by the obfuscators.

The detection ratio should not be very different irrespective of whether the malware was obfsucated or not.

But the results indicated that almost all the malware files had a very high probability of being classified as a benign file, if they had sufficient obfuscation techniques applied to them.

7.2.1 Metrics used

We use the detection ratio provided by VirusTotal to determine the effectiveness of the Malware Obfuscators. As expected, the malware detectors are not resilient enough to detect variants of malware that have been slightly obfuscated.

7.3 Obfuscation of Malware Samples

The results for applying each obfuscator were collected and only the significant results are shown here. In addition to gathering the results for an individual obfuscator, we also get the results for the individual malware detectors. A comparison of their behavior is also presented here.

7.3.1 Individual Obfuscators

Application of individual obfuscators did not alter the detection ratio by a huge margin. A sample detection ratio for applying the "Renaming" obfuscator is shown in Figure 7. As part of this experiment, 289 files were obfuscated and run through the malware detectors. In Figure 7, the malware samples are represented on the *x-axis* as alphabetic symbols. The mapping for the malware file to symbols is shown in Appendix B. The graph shows that the average detection ratio for the renaming obfuscator is 0.46. The average detection ratio is represented by a red line in the graph. While there are some occasional spikes in the detection rate, that seems to be the exception with a very few files being consistently classified as a malware.

The results for the obfuscator "Manifest" are shown in Figure 8.The detection

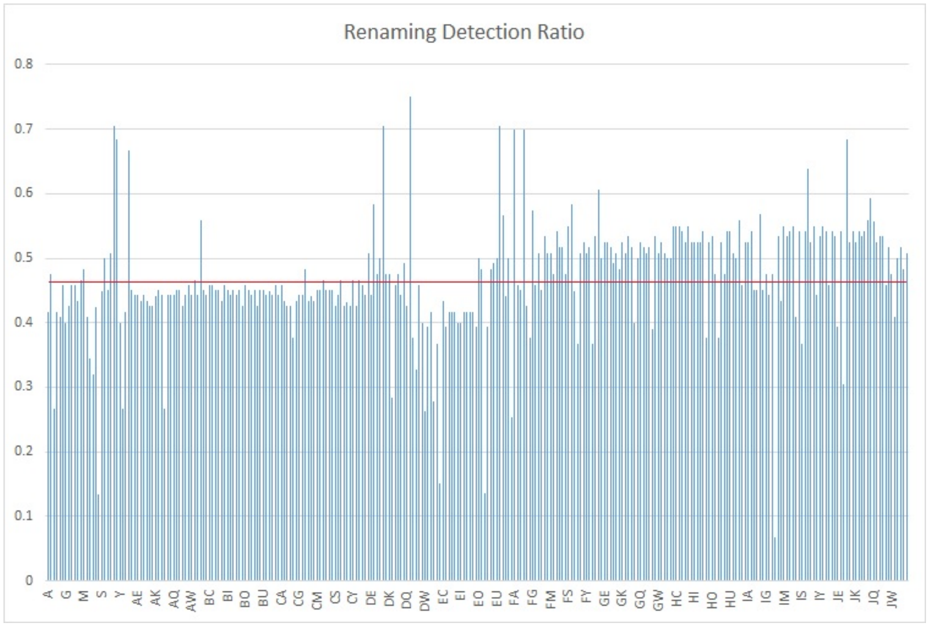


Figure 7: Single Obfuscator Usage - Renaming. Average: 0.46

ratio for this obfuscator is also lesser than for a normal malware. But with applying this obfuscator, the detection ratio is further reduced. The average detection ratio in this case is 0.3867. This shows that the manifest obfuscator contributes more to the detection rate. We repeat this experiment for different obfuscators to get record their detection scores. These are included in appendix C.

7.3.1.1 Multiple Obfuscators

While individual obfuscators didn't provide much insight into the malware detection scores, it was observed that combining multiple obfsucators quickly decreased the detection ratio.

In Figure 9, the obfuscators Renaming, Reordering, Goto, and Arithmetic Branching were applied to the files.

This certainly increased the obscurity of the malware files. The detection ratio for the obfuscated files in Figure 9 is much lesser than in Figure 7. This could be attributed to the fact that a combination of weak obfuscators is still a strong enough

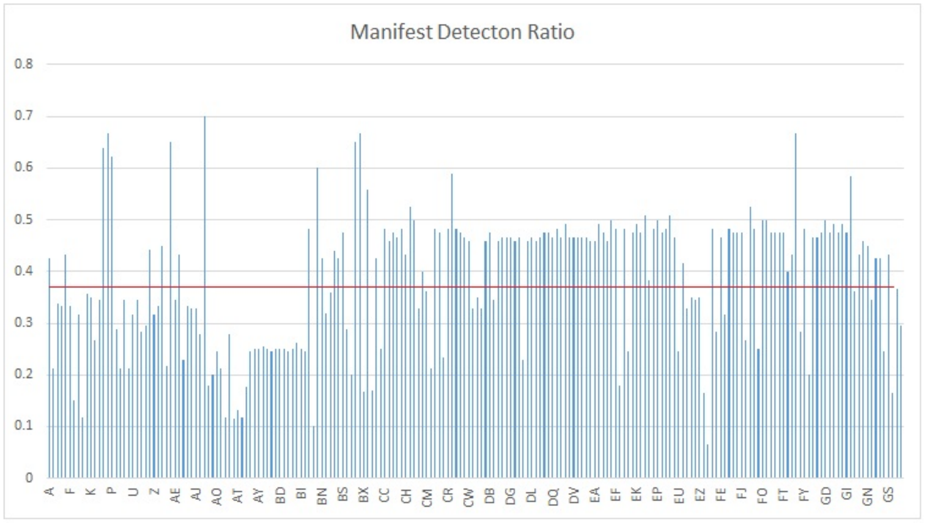


Figure 8: Single Obfuscator Usage - Manifest. Average: 0.3867

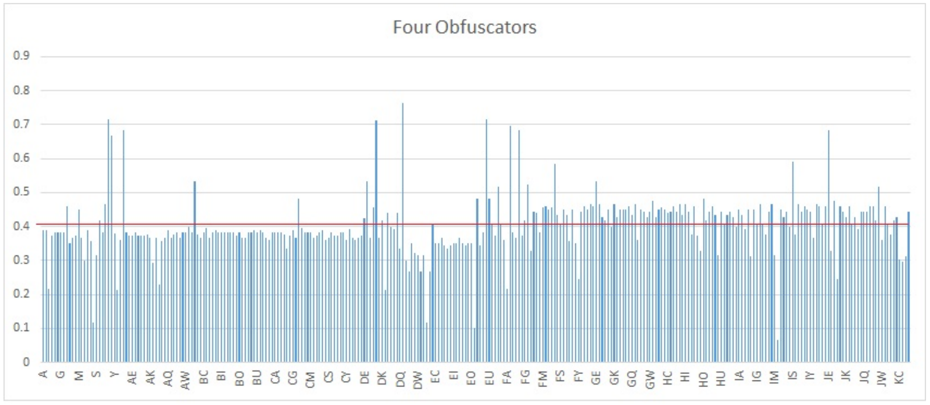


Figure 9: Detection Ratios with four obfuscators applied. Average:0.403457

challenge for malware detectors. We also note that, with a average of 0.4034, this is only marginally better than the performance of a single manifest obfuscator as shown in Figure 8. This further reiterates the significance of selecting the right obfuscator rather than a combination of different obfuscators.

7.3.1.2 All Obfsucators

To make the results of the experiment certain, all the obfuscators in question were applied to a set of files. Keeping up with the consistency observed so far, the

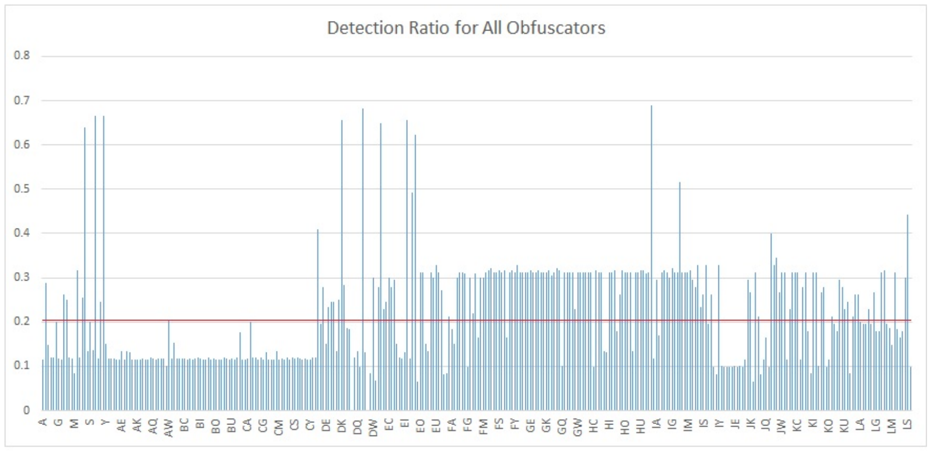


Figure 10: Pre- and Post-Obfuscation Results. Average: 0.219044

detection ratio dropped by a huge margin. This can be observed in Figure 10. The results show a average detection rate of 0.21. This is extremely low when compared to the results obtained before any obfuscation was applied as seen in Figure 6.

This clearly shows that by increasing the number of obfuscators being applied to a malware, we can bring down the detection ratio of that particular file to a very low value.

7.3.1.3 Average Ratio and Summarization

To conclude the experiments, the average detection ratio was calculated for each obfuscator and the combination of obfuscators. The results of this calculation are shown in Figure 11.

We observe a decline in the detection ratio for the different obfuscators. The average detection rate drops steeply when all obfuscators are combined. This is consistent with the results obtained so far in the experiment. The presence of the manifest obfuscator after the result of the combined four obfuscators in Figure 11 shows the importance of selecting the right obfuscator for defeating the malware detectors.

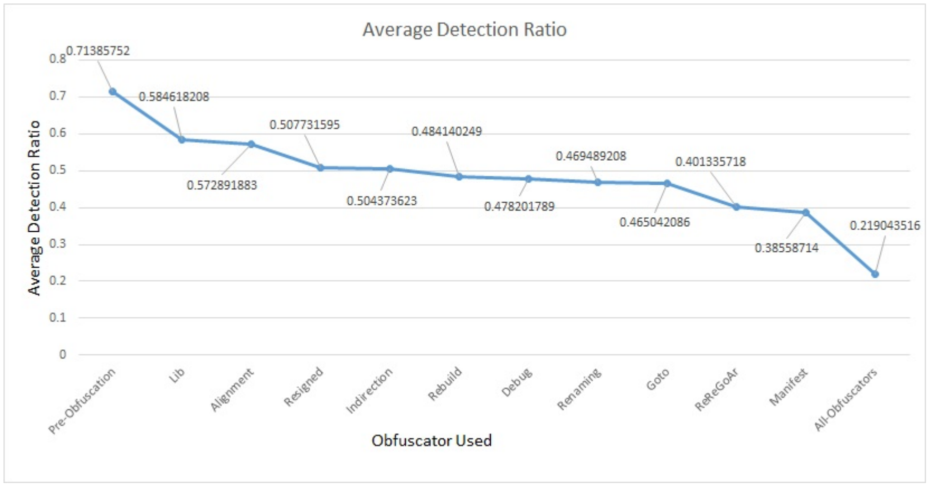


Figure 11: Average Detection Ratio for Different Obfuscators

7.3.2 Individual Malware Detectors

We now analyze behavior of the individual malware detectors and look at their performance against code obfuscation. To achieve this, we perform the same experiments as before and gather the detection statistics for each malware detector. This analysis will let us understand the workings of a particular malware detector and help us identify the best detector for Android. Once we have that information, we will know the best way to defeat obfuscation in malicious programs.

The detection rates for the AVG Antivirus are shown in Figure 12. From the figure, we can see that this Antivirus behaves in a manner that is consistent with the observations so far. The detection rates for AVG Antivirus are in line with the collective detection rates obtained for all the malware detectors. AVG performs very poorly only against the manifest obfuscator. In addition to this, the only other instance when this malware detector fares poorly is when all the obfuscators are combined.

Similar to the detection rates observed for the AVG antivirus, we look at the detection rates for the BitDefender, and the TrendMicro antivirus as well. The performance of the BitDefender Antivirus is showing in Figure 13.

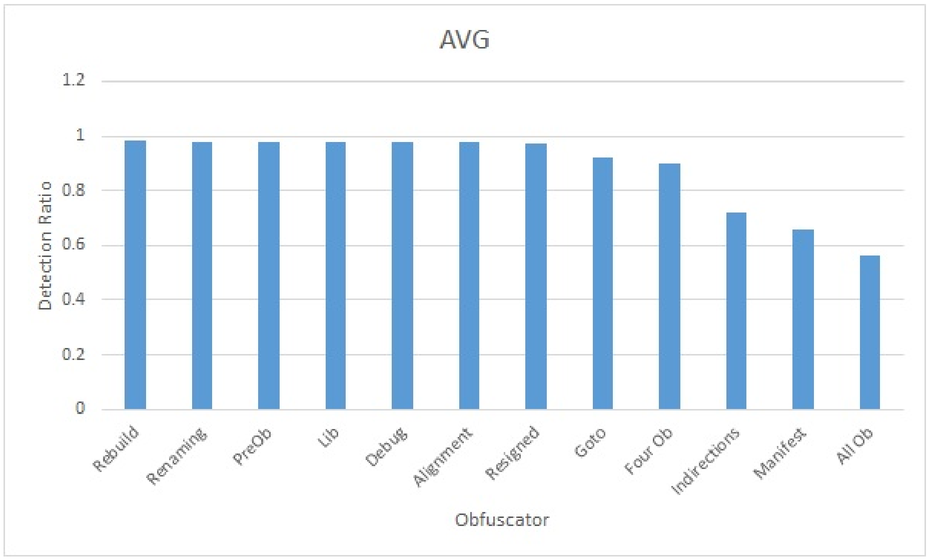


Figure 12: Detection Ratio for AVG Antivirus

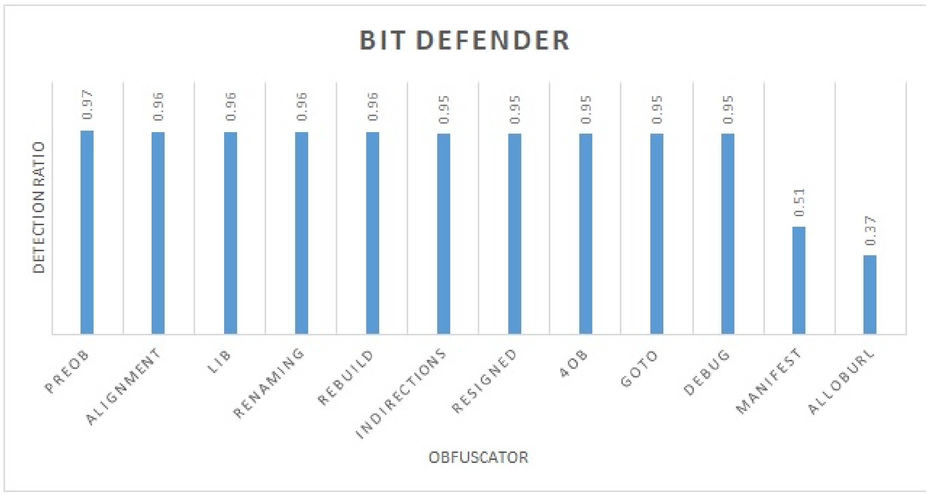


Figure 13: Detection Ratio for the BitDefender Antivirus

The BitDefender Antivirus program behaves in a manner that is consistent with the results obtained for all the malware detectors in general. The difference observed with AVG antivirus is not seen with BitDefender. However, it can be seen that the BitDefender antivirus still fails against the combination of all the obfuscators.

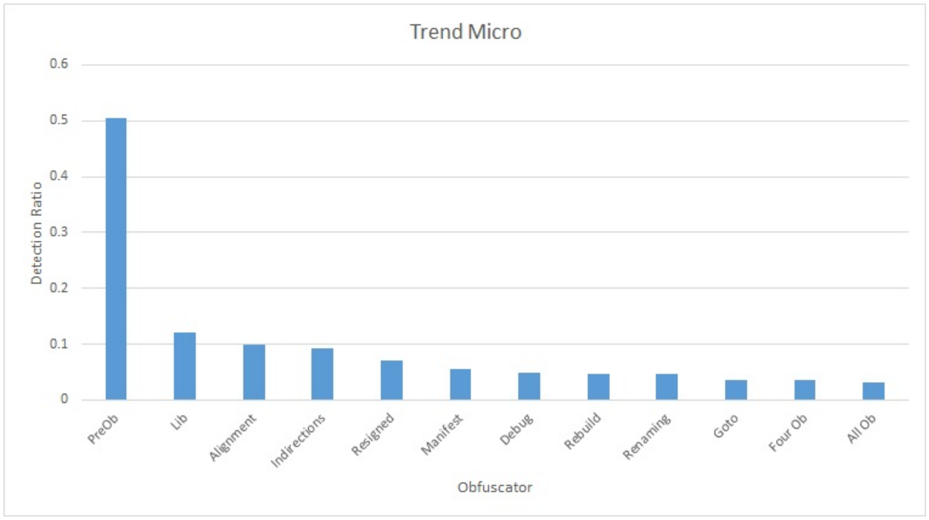


Figure 14: Detection Ratio for the TrendMicro Antivirus

The TrendMicro Antivirus performs well only against unobfuscated malware samples, as seen in Figure 14. The performance of this antivirus is consistently poor across all the obfuscators. We can surmise that the algorithm being employed by the TrendMicro antivirus is not very effective against obfuscated Android malware.

It is safe to conclude that the detection algorithms of AVG and BitDefender are much more efficient against obfuscated malware, than the algorithm used by the TrendMicro program.

CHAPTER 8

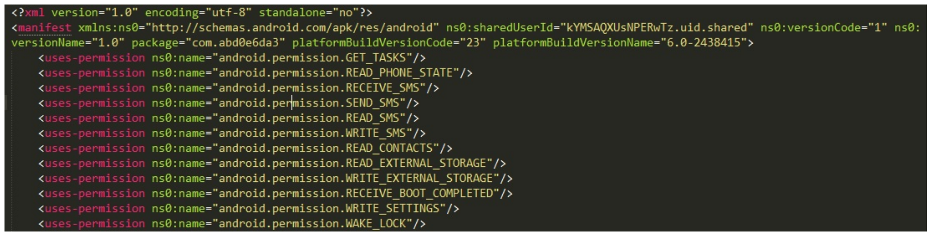
Conclusion and Future Work

From the results obtained in this experiment it is evident that the malware detectors of today are incapable of handling code obfuscation in android malware. This problem presents a huge gap in the domain of Android Anit-Virus products. The first step in building a robust malware detector for the Android operating system is to identify the flaws in the current implementation of the malware detectors. To be able to identify truly polymorphic malware, the anti virus programs need to be able to defeat the different types of obfuscators and their combinations. From this experiment, it is evident that the current malware detectors can be easily defeated and the only true defense against mobile malware is at the point of installation.

8.1 Conclusion

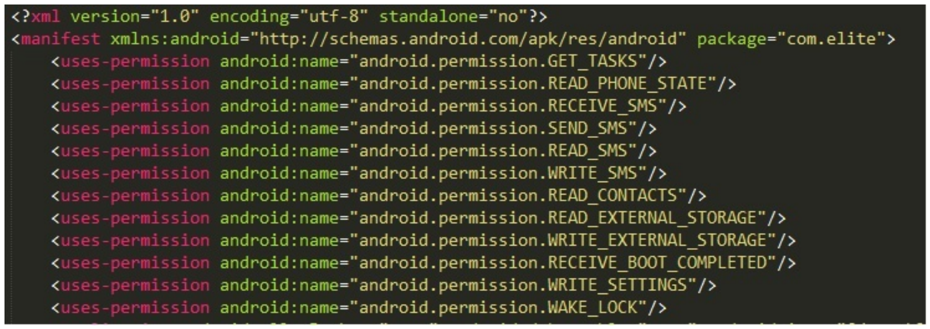
Due to the limited processing capacity of the mobile devices, it is imperative that stand alone malware detectors are able to sufficiently defend against known threats and variants of known malware that are detectable by signature scanning. In this experiment, we used different obfuscators to test the resilience of malware detectors against obfuscated malware. Unsurprisingly, the malware detectors fared very poorly against such obfuscation techniques.

We also observed that by applying all the obfuscators, it is becomes a trivial task to defeat a very large number of obfuscators. While certain obfuscators, such as the Lib obfuscator (explained in chapter 5 and results included in appendix C), do not contribute much to the detection mechanism, some other obfuscators contribute heavily to the detection algorithm. From the experiments, it is evident that the manifest obfuscator, as shown in Figure 8, contributes the most to the malware classification algorithm in most of the malware detectors. The importance of this



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:ns0="http://schemas.android.com/apk/res/android" ns0:sharedUserId="kYMSAQXUsNPERwTz.uid.shared" ns0:versionCode="1" ns0:
versionName="1.0" package="com.abd0e6da3" platformBuildVersionCode="23" platformBuildVersionName="6.0-2438415">
    <uses-permission ns0:name="android.permission.GET_TASKS"/>
    <uses-permission ns0:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission ns0:name="android.permission.RECEIVE_SMS"/>
    <uses-permission ns0:name="android.permission.SEND_SMS"/>
    <uses-permission ns0:name="android.permission.READ_SMS"/>
    <uses-permission ns0:name="android.permission.WRITE_SMS"/>
    <uses-permission ns0:name="android.permission.READ_CONTACTS"/>
    <uses-permission ns0:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission ns0:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission ns0:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission ns0:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission ns0:name="android.permission.WAKE_LOCK"/>
```

Figure 15: Part of an obfuscated Manifest file



```xml
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.elite">
    <uses-permission android:name="android.permission.GET_TASKS"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.WRITE_SMS"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
```

Figure 16: Part of a normal Manifest file

obfuscator can be gauged from the fact that this obfuscator gave better results than the combination of four other combined obfuscators 9. A part of an obfuscated sample manifest file is shown in Figure 15. A normal, unobfuscated sample of the same manifest file is shown in Figure 16. As can be observed, a simple switching of the values in the manifest file is enough to defeat the malware detectors. This leads to the conclusion that the selection of the obfuscators to use could greatly determine the detection chances for a malicious file.

Irrespective of the contribution of an individual obfuscator, combining the maximum number of obfuscators leads to significantly lower detection rates. Therefore, the higher the number of obfuscations employed, the lower the chance for a malware getting detected. The current generation of malware detectors are incapable of handling encryption in the body of malware. This experiment reiterates this fact and supports the conclusions drawn by Preda et al. in [17]. The conclusions drawn by them indicating a huge gap in the requirement and the availability of sophisticated anti-virus products is still very much prevalent.

8.2 Future Work

Similar to the obfuscators employed in this experiment, it should be possible to create "de-obfuscators" for Android files. It would be interesting to see the effect of each de-obfuscator against the corresponding obfuscator that has been used here. If employing such a de-obfuscator helps in thwarting the obfuscation, then it could form the basis for developing more generic de-obfuscation algorithms for incorporation into malware detectors. As was evident from the experiment, the selection of the right obfuscator could greatly influence the detection rate. This proves that the majority of the malware detectors place too much of significance on one aspect of a file, for classifying it. This shortcoming with the malware detectors for Android should be taken care of.

The detection mechanism employed in this experiment was employed various statistical methods. The experiment could be repeated with different datasets and with different dynamic detection methods. If dynamic detection mechanisms are able to defeat the obfuscators, they could be used to accumulate data over a large set of samples to create a library of known malware. This library could then be utilized by malware detectors that have very less processing power for performing dynamic analysis. The ease of decompiling and compiling APK files makes it an easy target for malware writers. If access to the source code of antivirus products are provided, defense mechanisms against such obfuscation techniques can be built in. With our increasing dependence on mobile phones and their proliferation into our lives, it is of utmost importance that sophisticated malware detectors are able to handle obfuscated malware.

With these conclusions, we hope to make future malware detectors more resilient against polymorphic virus with the expectation that the creators of the anti virus software incorporate the necessary changes to their programs.

LIST OF REFERENCES

[1] D.Emm, R.Unuchek, M.Garnaeva, A.Ivanov, D.Makrushin, and F.Sinitsyn, ‘‘It threat evolution in q2 2016,’’ 2016, accessed 2016-10-10.

[2] A.Apvrille and R.Nigam, ‘‘Obfuscation in android malware, and how to fight back,’’ *Virus Bulletin*, pp. 1--10, 2014.

[3] M.Christodorescu and S.Jha, ‘‘Testing malware detectors,’’ in *ACM SIGSOFT international symposium on Software testing and analysis*, New York, NY, USA, 2004, pp. 34--44.

[4] S. Gordon and R. Ford, ‘‘Real world anti-virus product reviews and evaluations--the current state of affairs,’’ in *Proceedings of the 1996 National Information Systems Security Conference*, 1996.

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, ‘‘On the (im) possibility of obfuscating programs,’’ in *Annual International Cryptology Conference.* Springer, 2001, pp. 1--18.

[6] V. Rastogi, Y. Chen, and X. Jiang, ‘‘Droidchameleon: evaluating android anti-malware against transformation attacks,’’ in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security.* ACM, 2013, pp. 329--334.

[7] Google, ‘‘Android security 2014 year in review,’’ https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf, accessed 2016-12-01.

[8] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, ‘‘An investigation of the android/badaccents malware which exploits a new android tapjacking attack,’’ Technical report, TU Darmstadt, Fraunhofer SIT and McAfee Mobile Research, Tech. Rep., 2015.

[9] Q. A. Chen, Z. Qian, and Z. M. Mao, ‘‘Peeking into your app without actually seeing it: Ui state inference and novel android attacks,’’ in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 1037--1052.

[10] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, ‘‘Androsimilar: robust statistical feature signature for android malware detection,’’ in *Proceedings of the 6th International Conference on Security of Information and Networks.* ACM, 2013, pp. 152--159.

[11] K. Ask, ‘‘Automatic malware signature generation,’’ Ph.D. dissertation, 2006.

[12] Statista, ‘‘Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2016,’’ https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, accessed 2016-12-01.

[13] Y. Zhou and X. Jiang, ‘‘Dissecting android malware: Characterization and evolution,’’ in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 95--109.

[14] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, ‘‘Android malware detection & protection: a survey,’’ *Int. J. Adv. Comput. Sci. Appl*, vol. 7, no. 2, pp. 463--475, 2016.

[15] R. Sato, D. Chiba, and S. Goto, ‘‘Detecting android malware by analyzing manifest files,’’ *Proceedings of the Asia-Pacific Advanced Network*, vol. 36, no. 23-31, p. 17, 2013.

[16] L.-K. Yan and H. Yin, ‘‘Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis.’’ in *USENIX security symposium*, 2012, pp. 569--584.

[17] M. D. Preda and F. Maggi, ‘‘Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology,’’ *Journal of Computer Virology and Hacking Techniques*, pp. 1--24, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11416-016-0282-2

[18] F. Pellegatta, F. Maggi, and M. D. Preda, ‘‘Aamo: Another android malware obfuscator,’’ https://github.com/necst/aamo, accessed 2017-02-17.

[19] R. Wisniewski and C. Tumbleson, ‘‘Apktool,’’ https://ibotpeaches.github.io/Apktool/, accessed 2016-10-26.

[20] ‘‘Contagio mobile malware mini dump,’’ http://contagiominidump.blogspot.com/, accessed 2016-12-15.

[21] ‘‘Virus total,’’ https://www.virustotal.com, accessed 2017-04-15.

APPENDIX A

Appendix 1

A.1 Abbreviations and Terminologies Used

- APK - A file format used for installing applications on the Android operating system.
- AAMO - Short for Another Android Malware Obfuscator - A program for applying obfuscators to android files.
- Obfuscator - A small program that scrambles the source code of a program to make its functions less obvious.
- Apktool - A program that is used to decompile and compile APK files.
- Smali -

    1. An assembler/disassemble for the dex format used by Android's virtual operating system.

    2. An intermediate file type between compiled dex code and java source code.

APPENDIX B

Appendix 2

B.1 Malware File to Symbol Mapping

Label	Filename
A	00d430877eed07d10c1e730926dcca9f82f282af
B	00e74c118fa3902e5c85fd8e37f3d084
C	08061663E638B5AC1D780CAACBE9FAD8_1074178_370436_GlamorousSmoke
D	0cbcfbebfb33fde66c282fec0248b0d99a829eab
E	0cc2c8461c78394b186a599c2d5baad364fb41c7
F	0D28FA54F9C0D41801E8FB5A7B0433DD
G	0e8236ddb163e7f3816cfef38b92c6e064887b3f
H	0ef158c897f91a58aa2a13d25cd3019bc19b9954
I	11A7767BFE4926458EC84385214B82C9
J	1485F498084F963801ED76013749C9FA
K	153c94a6d464497b07f1ea3511b87206a3621efd
L	156790b2ef37080cdc301324fa3f5a28d4c310d3
M	1F68ADDF38F63FE821B237BC7BAABB3D_IBanking_Chase
N	232e08bda4856b56e06a45ac5c27350fb30ddf5c
O	2C3B92FFE8123611AE9D9BED000C99F7_1074807_371216__3dtimeclockticks
P	2D66D7942148DE2D9F08EAB403921C89
Q	314d66e71040b36ba63ad5a376647dd63ecf3a5c
R	3E076979644672A0EF750A4C3226F553_assassins_creed
S	4021A1E00B3ABEE730994F1EE17219B4
T	4084939A0864B645F6C6A915586FB1AB_com.gmdcd.pic_1345165918398
U	40F3F16742CD8AC8598BF859A23AC290
V	4A300481411AB1992467959491DF412C
W	4D13D1BC63026B9C26C7CD4946B1BAE0_com.bntsxdn.pic
X	4d3a1a769255402be23ae5e6b3445d79b7b4b702
Y	4e80480daf4ab573121d839c2c74cc845945be38
Z	4FD1194F8127439609319CDBE244C0A7_1074349_370686__BlueArt
AA	55d716895ea0934c4a91e1e2cfbd682dec30cb2f
AB	55e2a4d0d89bc70e84159385ed9f078c5d7d9947
AC	561b37c04e92e1a4aadbc51138c787863408a014
AD	564431a34d65836481741ed83d6cb21c9a9bb7ba
AE	56b70b6d31dc3315cdd3b448416f2e2704a1ab25
AF	574e59a377b696c4bdfb83d4bef5478891c000e0

AG	57e8901381a4e9de94b26f458499c49051b19af2
AH	57f21111f6da9fb9a18af88dff688e59e8e24156
AI	58E73A03025BA95337C952223F18F479_1074703_371102__lordssacredheav enlycross
AJ	598df80d1d5279e3204ef023dd4dbbe08be6bbd9
AK	5a37e9dd95ffaaae0c29197d2b45fd2afdf77f05
AL	5af738a737ce7ab4005505ab9ca43b08d4e3b503
AM	5c325c70250cbd294fae4cb321b3d8d39f1c1cd3
AN	5c53c9e54294250c0318c35086523449fa917f5c
AO	5cd906b76a1c15373bc7a0ed0d24ef69f84b2c28
AP	5cddd6f6585b0dff93ce1ecc6d8680e83c61e5b3
AQ	5d42e63a02548c15801c2da5b16cbcfb33c4230b
AR	5e9a4e1bb7fb4c94bceef4cd2af54bddaf1f1c34
AS	5f0b8bb59061451a5e45241858c3f8ac62569371
AT	5f0ba094e83ee321b331a3acd7252ae92b4d5734
AU	5fdcb3d86a949d73ddbf721640733917dc300d41
AV	60761527bdec07e7cf5fc35c8aaccf4de7617649
AW	60b1c98fc6ca2b86fbd7c772dc08a73e
AX	60b4ef7037ca6a4d1ee7e3c35c8e27d7
AY	6107f1f26bcd78b628f80e4531998c4b9444ca77
AZ	613398fef32a47a195ae493c8e635ceab6f4fcbd
BA	6214285ed81d3209d4947efe3a2291034877d417
BB	6260c6ba44308c0c4610468784b055ad69fa1095
BC	62bf7ab29610d47737ce01b9becbf4f56651e367
BD	62f6d3b57f0bcea6b9edebff7d67b4a1fb7ece7d
BE	634283bcea6d075b157b76a5f88d23cee733fcb7
BF	63616b5ed2253761c3e9aa47bc155a1743ac9a6f
BG	637d93c7c4d63b5c5d292c24a4a3ddff0f89cb99
BH	6386ea80441002cbfd69fd8ab74b7921d4378abb
BI	63e46c5c180d9b83a5866e770df00cadcc746e6a
BJ	63fc9581928251540df5a811eb20b9024065fcc9
BK	6414962b8bdc09247d92c1317a3e0aa31a973de2
BL	64a8be553cd05c4ac08738df819f231fc16b4b6c
BM	65324abd9ceb8166487d756f474c04ab618b5c30
BN	654d374da14a9edb95f85651be60e1888f237b98
BO	6594767af663113e6c46d2a3ede5d87ec1d034ee
BP	6599cffb03d95b07dafe8e1be726b160d7541c33
BQ	65d40b7b0e9eda5d5a209f3d34ed93357289dafe
BR	65f66e7b862db8c23074da1c2fe697d594ca1cdc

BS	661cc12f341af0120fbe74b33a8bc4863cae37b9
BT	667a3d0763101b1494c981fbdb9f6f18a41ecabc
BU	669f41369d3bfa56439e7fb6ef01a4a36e08729c
BV	6726709a16a54d457a8d4da73cc45bc5295d7168
BW	676d73270dfd198a8d7867e1df243dbb9b0e102e
BX	69B9691A8274A17CDC22E9681B3E1C74
BY	69be497da755a8259af5cdeda4ac0c9de67a81e2
BZ	6a6176fc043b821b1ceb48425f2bce9c1f3a6cb8
CA	6b26dd8548bad85e2b4bbf2650dc3c5879abc029
CB	6BAE149BC65576831AC635A23938BE36_smartphone5-1
CC	6bb6b3143790f0870f39e80cd3d6bd78fb3a9a57
CD	6c0b900a17faf11d9efc68951b2d04fdb180bfe8
CE	6c13a359586f9cab20f2bc9b4fd8294e61e6e852
CF	6c93ef2106647eb9e9322de5d106ae9df6146277
CG	6d02439c416349545211e382bc0f27b2383123f1
CH	6d43b3bc85770fafeb598eb5297bc341
CI	6d6b779ea0b3d31c9453db8268b1e85463fe4725
CJ	6db96e8a52382fa6f2d3220b592d7ae92f1d78f2
CK	6dba2c4cc420d3c43067cd0f8a86e1718f9639cb
CL	6dbef6bf711c74227550da5a033a0ae4c4c1c1cb
CM	727a33c78e4329ee5e1586a13ee867132790e436
CN	737395cf1bccbc23531fb109b4a8ee1e8cce26b4
CO	73ff558ea62c0835761eced6b292cc930728cf43
CP	74333980ae5bafcb25a9031fb46275435cdbba2e
CQ	749ff6f09b3b6de044ddadf447860b7fd63d8672
CR	74d9dc5a2c95e9eaa880ec11a32d9b109794474b
CS	75459a5009bf08067a1e15ee4e2992c23e00433c
CT	75f31fe1a07986080b6a6f4cd2d9347cc72201b4
CU	761c6c36d81c1edd9e0645447a4e638d7d88356e
CV	766a65fe6d1e4be4551d7d30a1b4539f19991e0e
CW	76f3739c16fb978eafde4ebfae105dc8a94731a5
CX	780b5f7c07ab98de7d8d07eed781973a415ebc5d
CY	780d5124b448249d948a60b43775a424634024ac
CZ	7828066c4804b6364a6f55b6aff3b657899a9d99
DA	792BBB3DDC46E3D0E640D32977434ACA
DB	7c0e0b1ca01e97c2f0d043eb0aabe61cae6216f7
DC	857ee29d88796e1f1b7b440dc9eadc77
DD	88870ad3c7bd42cfe1d728b4a4ccc104
DE	8D52070201F2A81FB1298E133D74057C
DF	8d574d94ba9445979723cfc810637fd84d4c06e1

DG	8F7A41A921FC15F4FD47A33E476D7B3B_1074179_370437__SkullLighter
DH	9C9AFD6B77D8D3A66A2DB2D2CF0B94B3
DI	9d1625aa79b55a79064dac7a0ecc2f91
DJ	A31245022C60FC50B81F7FFC4F4967B2_com.hxmv696.pic
DK	A4D6033F66DA3BE83CBF80724CA013D1
DL	Activator
DM	alfasafe
DN	Alsalah
DO	Android.Beita_com.beita.contact_10953B741D166D9E22937FE00FBF1038
DP	Android.Core.Defender
DQ	android.dds.com-STiNiTER
DR	Android.Hehe_1CAA31272DAABB43180E079BCA5E23C1
DS	Angry_BirdTransformers_1.1.0
DT	apk
DU	atticlab.bodyscanner
DV	B0E22A785041229A644F015472E738BA_1074810_371221__ghostiderfirefla messremixFAMOUS3DAPPS
DW	B2B7D5999DCE0559D13AB06D30C2C6EC
DX	B6CACC0CF7BAD179D6BDE68F5C013E6E_xqxmn18
DY	B8B434AB21D394DAA0A9A78A515BD517
DZ	b9622e587ae28cfff8ffc5645221e422
EA	BlackList_Pro_v2.8
EB	btm
EC	c1f9283b7ad8457160d3c189430f2c75
ED	c2dfe44d9f130033ecd89ba33f8a2e0a
EE	C424F9AD311F3B55F8DB5DABF6985856_Accutrack
EF	C71740EE94467AE70A71265116D54186_com.zqbb1221.pic
EG	c85d37585dbe2ad77572d9a27165ed63c9c8685e
EH	caa04deff90081fd4b0b441b9bf16edeb05f52ee
EI	CAFFFDEE7479A8816F4551AC8C3A0178
EJ	carddeemamaAndroid
EK	CCC01FD6D875B95E2AF5F270AAF8E842.576B9B86
EL	cce1a35b5fee30883ea3ddca8312109691116cba
EM	CE7B9B2242A71BBEAC0B2839B1063013_1074139_370393__NoiseDetecto rNonG
EN	cenix.android.vbr
EO	CFB7E66B2FB605CC94DEBD01238B4995
EP	ch.smalltech.ledflashlight.free

EQ	com.adobe.air
ER	com.adobe.flashplayer
ES	com.adobe.reader
ET	com.advancedprocessmanager
EU	com.alioth.imdevil_jp.DevilsCreed.full_1.8_installer
EV	com.android.googledalvik
EW	com.android.googlekernel
EX	com.android.installer.full
EY	com.android.locker
EZ	com.android.Materialflow
FA	com.anglefish.livewallpaper.hotchick1
FB	com.antivirus
FC	com software.compass
FD	com.app.lotte.auth-1
FE	com.appspot.swisscodemonkeys.jokes
FF	com.appspot.swisscodemonkeys.paintfx
FG	com.atools.cuttherope-LeNa.b
FH	com.bb.iphone
FI	com.biggu.shopsavvy
FJ	com.c101421042723
FK	com.cootek.smartinputv5
FL	com.devuni.flashlight
FM	com.droidmojo.awesomejokes
FN	com.dropbox.android
FO	com.ebay.mobile
FP	com.estrongs.android.pop
FQ	com.evernote.skitch
FR	com.facebook.katana
FS	com.facebook.orca
FT	com.fdhgkjhrtjkjbx.model
FU	com.fede.launcher
FV	com.gau.go.launcherex
FW	com.gau.go.launcherex.gowidget.taskmanager
FX	com.gau.go.launcherex.theme.iphoneazooz
FY	com.google.android.apps.maps
FZ	com.google.android.apps.plus
GA	com.google.android.apps.translate
GB	com.google.android.stardroid
GC	com.google.android.street
GD	com.google.android.voicesearch

GE	com.google.android.youtube
GF	com.google.earth
GG	com.google.zxing.client.android
GH	com.hm
GI	com.icq.mobile.client
GJ	com.incredibleapp.wallpapershd
GK	com.intsig.camscanner
GL	com.jb.gosms
GM	com.jiubang.goscreenlock
GN	com.lovekamasutra.ikamasutralite
GO	com.metago.astro
GP	com.movieshow.down
GQ	com.mxtech.videoplayer.ad
GR	com.netbiscuits.kicker
GS	com.nnew.GTAHDBackground
GT	com.opera.browser
GU	com.opera.mini.android
GV	com.outfit7.talkingben
GW	com.outfit7.talkinggina
GX	com.outfit7.talkingsantafree
GY	com.outfit7.talkingtom
GZ	com.parental.control.v4
HA	com.piviandco.fatbooth
HB	com.qq.assistant
HC	com.rechild.advancedtaskkiller
HD	com.saavn.android
HE	com.sancronringtones.funnysmssb
HF	com.security.patch
HG	com.skype.raider
HH	com.splunchy.android.alarmclock
HI	com.starfinanz.smob.android.sfinanzstatus
HJ	com.stephbriggs5.batteryimprove-2
HK	com.stylem.wallpapers
HL	com.teamviewer.teamviewer.market.mobile
HM	_com.tebs3.cuttherope_6_1.1.5
HN	com.viber.voip
HO	com.vlcdirect.vlcdirect
HP	com.vlingo.client
HQ	com.VoiceChange.VoiceChangeIL-1.4
HR	com.watchtv

HS	com.wetter.androidclient
HT	com.whatsapp
HU	com.whatsapp.wallpaper
HV	copy9_23
HW	ctm
HX	D67A07E3DE88C0130420588FD158B967_1074808_371217__eyeseeyouSAMSUNG
HY	d
HZ	DE5BFA8715DAC2E29E206C19CA98F2F4_1074141_370394__JingleBellNonG
IA	de.blitzer
IB	de.cellular.tagesschau
IC	de.dasoertliche.android
ID	de.frauentausch.andreas
IE	de.hafas.android.db
IF	de.is24.android
IG	de.kaufda.android
IH	de.mehrmannd.sdbooster-GAMEX
II	de.spiegel.android.app.spon
IJ	de.tvspielfilm
IK	de.web.mobile.android.mail
IL	de.wutprobe211.de
IM	dtm
IN	E1B86054468D6AC1274188C0C579CCAF_iBanking
IO	E8063DE12976D371441F15F2C5715627
IP	e8237a583fe7b2362b4addf01518600b
IQ	Extension.2nd.stage
IR	F05839EB7156B434A893BBEDDB68AD85
IS	F06AF629D33F17938849F822930AE428_iBanking_ing
IT	F1AA24C1641471F5FBEF08AE56A53FB4
IU	F1BC8520754D2AC4A920B3EF5C732380_iBanking_bot
IV	F836F5C6267F13BF9F6109A6B8D79175_fbi
IW	fakeAV_148B76C664F2854E2947AF01160FFA99_LabelReader
IX	fakeAV_1CA532F171A0B765A46AF995EBAAB1D2_LabelReader
IY	fakeAV_1E178E501B41659FFACE85153615DEA7_LabelReader
IZ	fakeAV_36B177910C99872B33E90DEA71B16617_LabelReader
JA	fakeAV_6F237D25472D9D09FC44ECE7DC9CED92_LabelReader
JB	fakeAV_75B8F9DBB1CD79B7FC074F7F499150CF_LabelReader

JC	fakeAV_77BB7F86FB0AC66C97B1AB3573ADFFC1_LabelReader
JD	fakeAV_934527F8EBB5C1088009CC9329DC3DE6_LabelReader
JE	FAKEAV
JF	fakeAV_ED1E0689F93B0C57E403489BB5338F59_LabelReader
JG	Fakemart_.D002F0581A862373AA6C6C0070EC3156
JH	FakeSMSInstaller_Geared_1.0.2
JI	FB9FEFFB1FEF13C4A5E42ACE20183912_1074813_371228__SaveTenDollar
JJ	flashplayer.android.update
JK	GoogleKernel
JL	hippo_sample
JM	HtcLoggers
JN	hu.tonuzaba.android
JO	il.co.egv-3
JP	instagram
JQ	install
JR	jin_old_2.1
JS	kim
JT	krep.itmtd.ywtjexf-1_02E231F85558F37DA6802142440736F6
JU	kr.sira.measure
JV	kr.sira.sound
JW	la.droid.qr
JX	live.photo.savanna
JY	Loozfon_04C9E05D0F626CC3F47DC0BC9B65A8CF
JZ	miyowa.android.microsoft.wlm
KA	mms475843
KB	net.uloops.android
KC	net.zedge.android
KD	Newfpwap_com_liveprintslivewallpaper
KE	org.leo.android.dict
KF	org.mozilla.firefox
KG	PhoneLocator_Pro_4.6
KH	_pl.byq.new_19_1.2.5
KI	Ransomware-locker-67BDE6039310B4BB9CCD9FCF2A721A45_koler
KJ	ru.blogspot.playsib.savageknife
KK	SandroRat
KL	santander
KM	sb
KN	sber
KO	Scan-For-Viruses-Now
KP	schgg

KQ	smart.apps.droidcleaner-1
KR	smart.apps.superclean-1
KS	smtp_C9B7BE2C1518933950B0284FC254C485_20130802_031615
KT	sp_k_test
KU	sp_mtm
KV	sp_ntm
KW	spyera
KX	SuiConFo
KY	SuperClean-11
KZ	suspect
LB	test97
LC	test98
LD	test99
LE	testService
LF	ThreatJapan_4C937667CB23E857D42B664334E1142A_NewsAndroidcode03
LG	ThreatJapan_BA73E96CAA95999321C1CDD766BDF58B_NewsAndroidcode02
LH	ThreatJapan_CF45E1288B47D97326ED279F2EE41E4D_NewsAndroidcode01
LI	ThreatJapan_D09A1FF8A96A6633B3B285F530E2D430_NewsAndroidnocode
LJ	tunein.player
LK	tvtotalnippeltrial.app
LL	uk.co.exelentia.wikipedia
LM	Update
LN	vertu.jp
LO	vertu.kr
LP	vksafe
LQ	waterfall3dLive.boa.liveWPcube
LR	Whats_app
LS	XXshenqi
LT	zitmo

APPENDIX C

Appendix C

C.1 Single Obfuscator Results

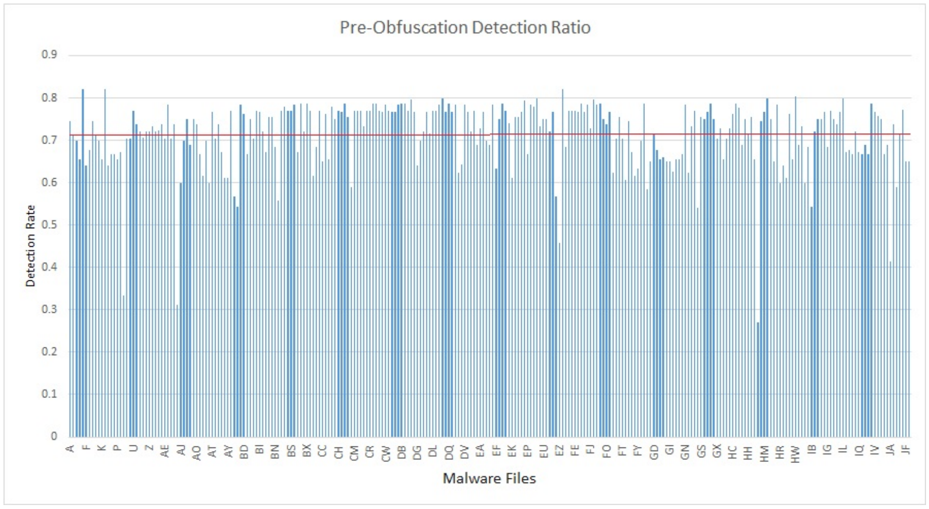


Figure C.17: Detection Rates before obfuscation. Average: 0.7138

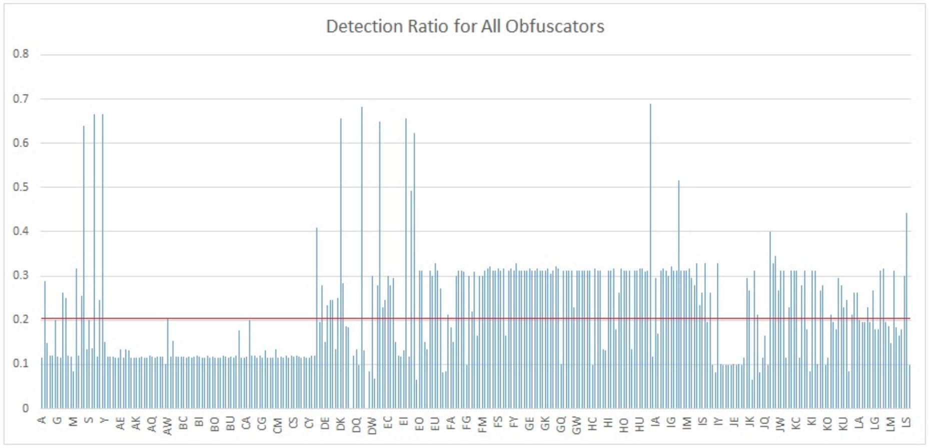


Figure C.18: Average Detection Ratio after using All Obfuscators. Average: 0.219044

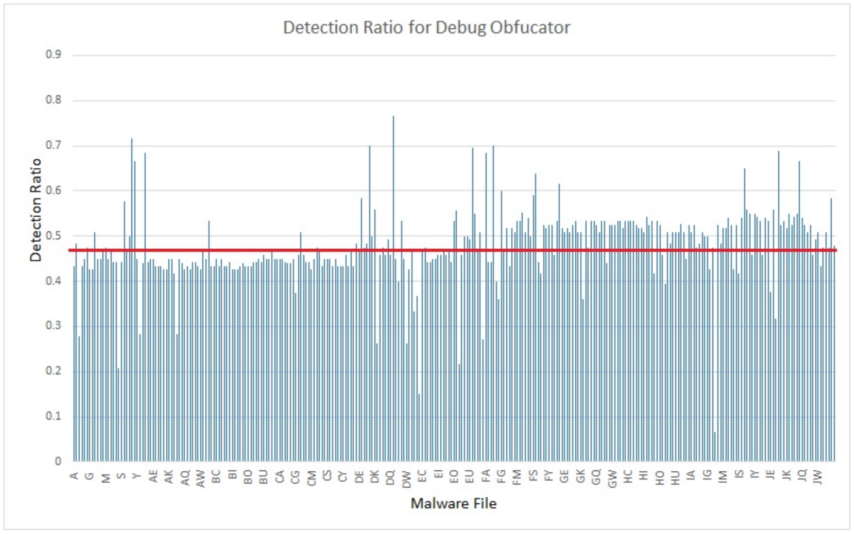


Figure C.19: Detection Rates after applying the Obfuscator Debug. Average: 0.478202

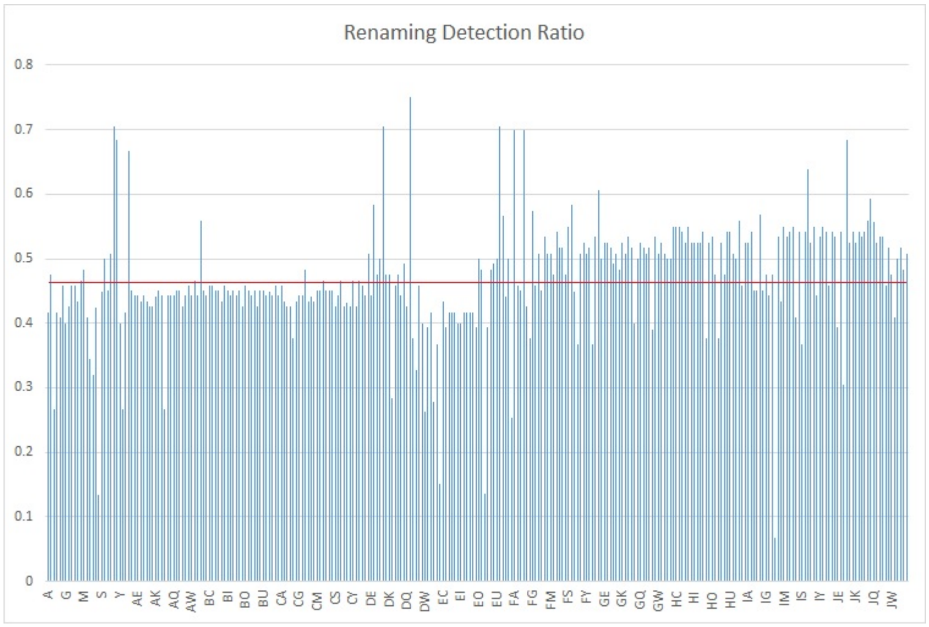


Figure C.20: Detection Rates after applying the Obfuscator Renaming. Average: 0.478202

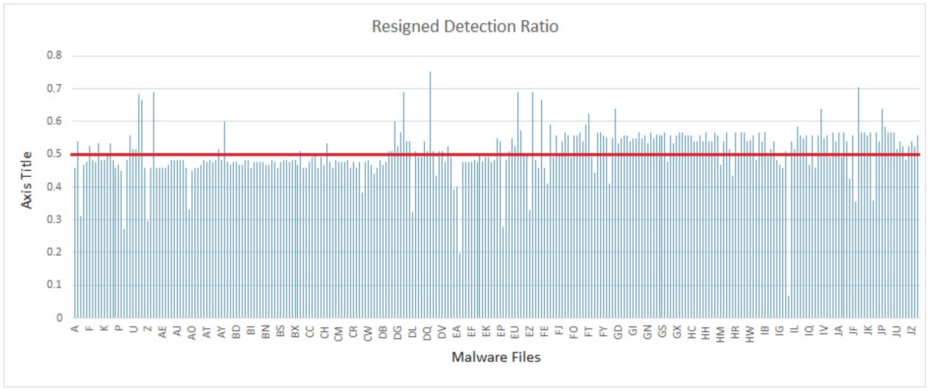


Figure C.21: Detection Rates after applying the Obfuscator Resigned. Average: 0.507732

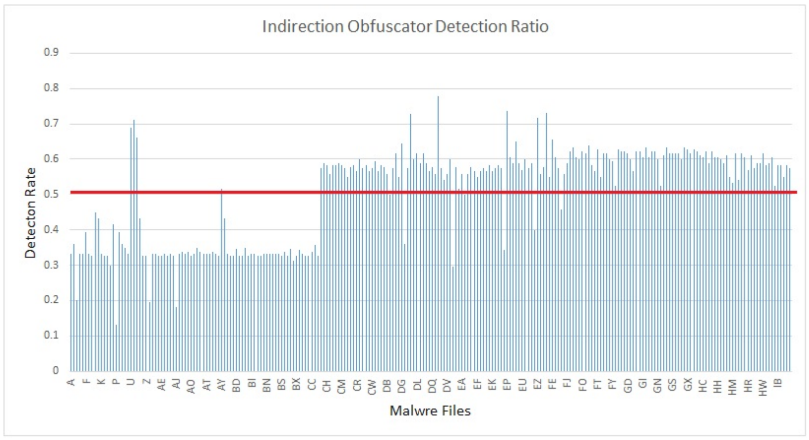


Figure C.22: Detection Rates after applying the Obfuscator Indirection. Average: 0.504374

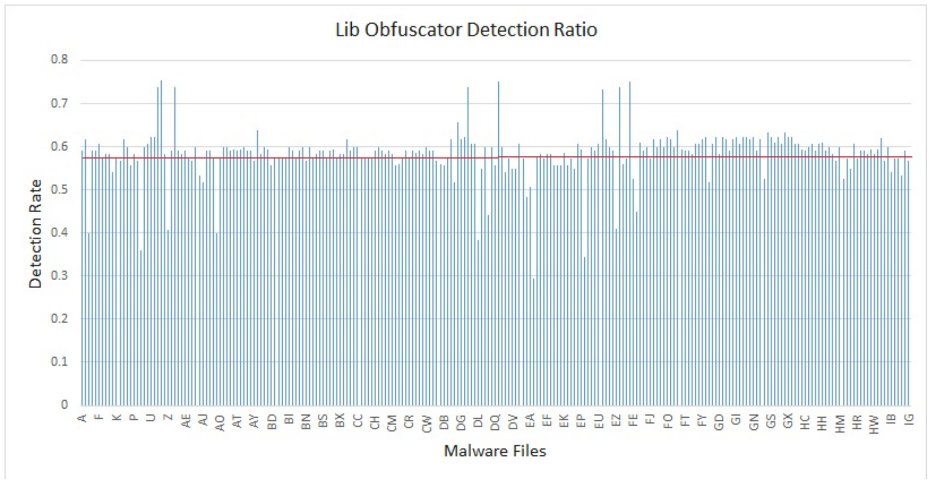


Figure C.23: Detection Rates after applying the Obfuscator Lib. Average: 0.584618

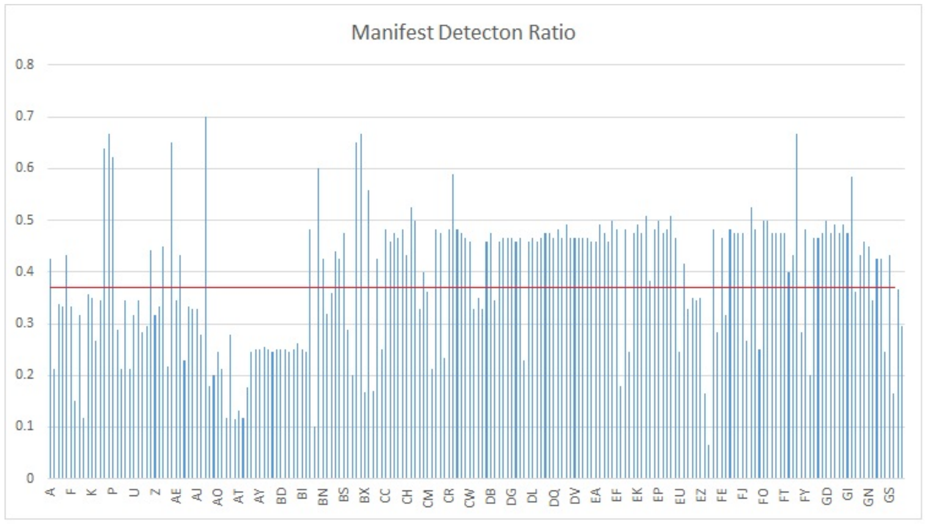


Figure C.24: Detection Rates after applying the Obfuscator Manifest. Average: 0.386791

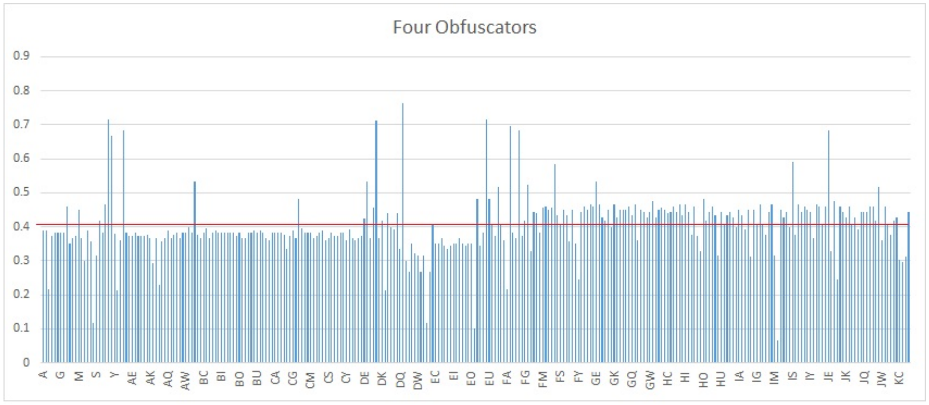


Figure C.25: Detection Rates after applying the Obfuscator Renaming, Reordering, Goto, and Arithmetic Branch. Average: 0.403457

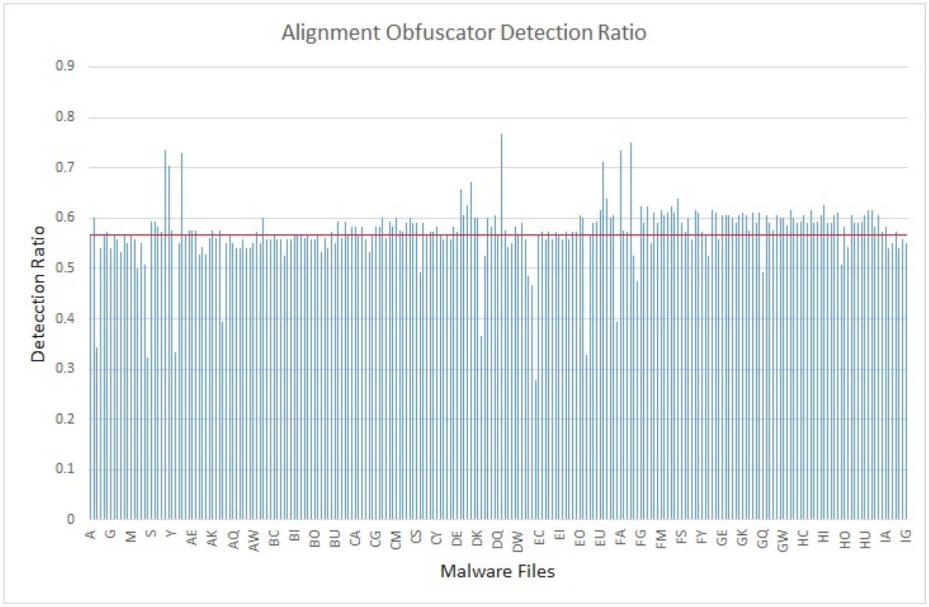


Figure C.26: Detection Rates after applying the Obfuscator Alignment. Average: 0.572892