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

Dr. Mark Stamp	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science

ABSTRACT

Black Box Analysis of Android Malware Detectors

by Guruswamy Nellaivadivelu

One of the challenges of combating mobile malware in Android devices is attributed to code obfuscation. Malware writers obfuscate the source code of their programs by employing various techniques that attempt to hide the true intent of the program. Malware detectors use a handful of features to classify a program as a malware. If the malware detector uses a feature that is obfuscated, then the malware detector will fail to classify it as a malicious software. In this survey, we look at works of literature that discuss code obfuscation in malware and the challenges faced by malware detectors to combat the code obfuscation.

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 grateful to my committee members for reviewing my work and providing constructive feedback.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Android malware proliferation is rising exponentially. In the second quarter of 2016, 3.5 million 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 malware. One of the challenges faced in this area is the use of code obfuscation techniques. Code obfuscation is a method of altering a source code to hide its actual purpose. There are many ways of obfuscating a 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]:

1. Question 1: How resistant is a malware detector to obfuscations or variants of known malware?

2. Question 2: Can using limitations of a malware detector in handling obfuscations determine its detection algorithm?

The two questions above can be used as a way to gauge how good a malware detector will perform against obfuscated code.

1.1 Code Obfuscation

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.

1.2 Challenges

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 more sophisticated tools for the encryption of source code for malware files, it is becoming increasingly difficult to differentiate instances where obfuscation is used for a genuine security reason and instances where it is being used with a malicious intent.

1.3 Objective

The primary objective of this project is to make malware detectors more responsive to the code obfuscation techniques employed by malware writers. The objectives can be broken down into the following two points:

1. 1.Identify the malware features that are used by a malware detector by encrypting it.

2. 2.Modify an existing malware detector to overcome the limitations of code

obfuscation.

The first step in 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.

CHAPTER 2

Literature Survey

2.1 Introduction

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.

2.1.1 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 [3]. 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.'

2.1.2 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.

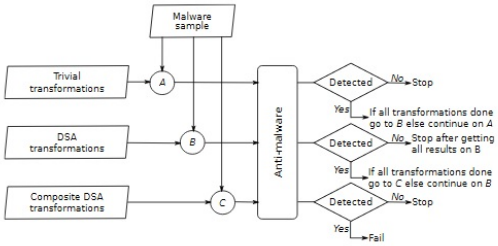


Figure 1: Evaluating anti-malware

As shown in Fig. 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 transformation, the anti-malwareâĂŹs 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.

2.2 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 âĂIJTapjackingâĂİ 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.

2.2.1 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 that takes into consideration

various features that are known to be present in malware alone. The AndroSimiar approach, as shown in Fig. 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.

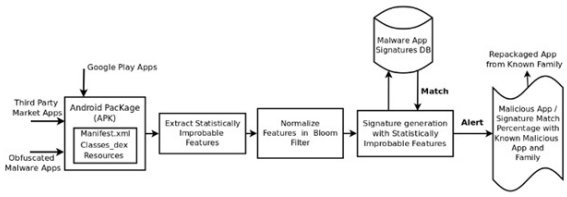


Figure 2: AndroSimilar

2.3 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.

Some of the advantages of code obfuscation, that make it very popular amongst software developers are:

- Prevention of reverse engineering.
- Protection of intellectual property.
- Reducing the size of an executable.
- Protection of licensing mechanisms.
- Restricting unauthorized access.

We will look at the history of code obfuscation to appreciate the relevance of code obfuscation in today's software development perspective.

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 B̈rain 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 statical 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. The various methods employed by antivirus programs include:

1. Signature Based Detection
2. Heuristics Based Detection
3. Rootkit Detection

4. On-Access Scanning

The functioning of theses methods, their limitations, and their relevance to this experiment are discussed in the next few pages.

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

Android Malware Obfuscators

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 fig. 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

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. As shown in the fig. 4 obtained from [13], we can see that benign and malicious Android applications have access requests that are very similar.

Due to this, using access requests as a measure for classifying android applications became ineffective. The various types of Android malware can be broadly classified into the following categories, depending on the type of malicious activities they perform [13]:

- Privilege Escalation

- Remote Control
- Monetary Loss
- Information Collection

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. 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.

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

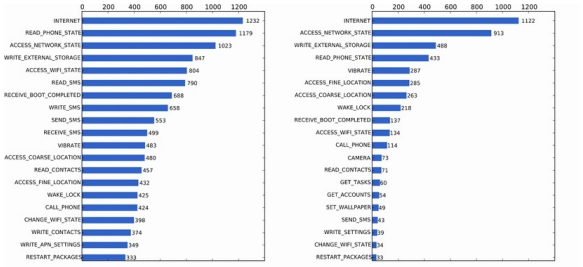
In this method, the application is executed and it is analyzed during the runtime. It becomes very easy to identify sections of code or execution blocks that were missed during the static analysis of an application. Dynamic analysis methods are also effective against obfuscation and encryption techniques.

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.



(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

CHAPTER 5

Obfuscators in Android Malware

In this experiment 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.

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 Fig. 5.

As shown in Fig.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 the following 14 obfuscators to test out the resilience of the malware detectors:

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.

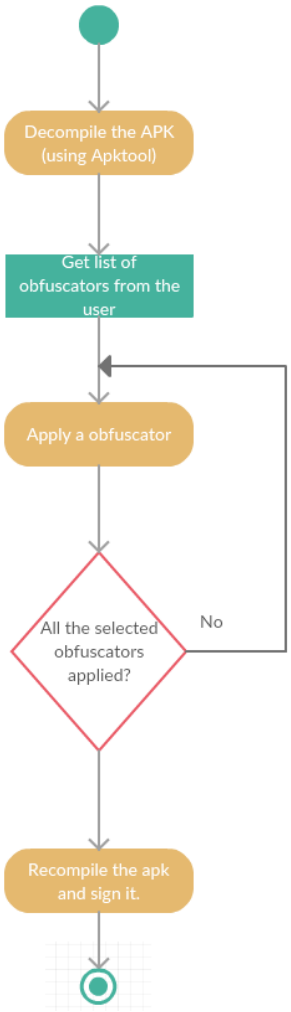


Figure 5: Experiment flow

CHAPTER 6

Software Requirements and 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 Fig.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 1.

Table 1: List of software required and their versions.

S.No.	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.

LIST OF REFERENCES

[1] M. A. D. F. D.Emm, R.Unuchek, ''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://static.googleusercontent.com/media/source.android.com/en//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://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_2014_Report_Final.pdf, 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.