EVALUATION OF OBFUSCATED ANDROID MALWARE

Co-authored by

Himanshu Patel, Deep Patel, Jaspreet Ahluwalia,

Vaishali Kapoor, Karthik Narasimhan, Harmanpreet Singh,

Harmanjot, Gadi Harshitha Reddy, Sai Sushma Peruboina

Research Project

Submitted to the Faculty of Graduate Studies,

Concordia University of Edmonton

In Partial Fulfillment of the Requirements for the

Final Research Project for the Degree

MASTER OF INFORMATION SYSTEMS SECURITY MANAGEMENT

Concordia University of Edmonton

FACULTY OF GRADUATE STUDIES

Edmonton, Alberta

Advisor: Dr Sergey Butakov (sergey.butakov@concordia.ab.ca)

June 2021

EVALUATION OF OBFUSCATED ANDROID MALWARE

Himanshu Patel, Deep Patel, Jaspreet Ahluwalia,

Vaishali Kapoor, Karthik Narasimhan, Harmanpreet Singh,

Harmanjot, Gadi Harshitha Reddy, Sai Sushma Peruboina

Approved:

Sergey Butakov [Original Approval on File]

Sergey Butakov Date: June 23, 2021

Primary Supervisor

Patrick Kamau [Original Approval on File]

Patrick Kamau, PhD, MCIC, PChem. Date: June 23, 2021

Dean, Faculty of Graduate Studies

Table of Contents

List of Tables

List of Figures

Evaluation of Obfuscated Android Malware

Himanshu Patel
hcpatel@student.concordia.ab.ca

Vaishali Kapoor
vkapoor@student.concordia.ab.ca

Harmanjot
hkaur19@student.concordia.ab.ca

Deep Patel
dpatel6@student.concordia.ab.ca

Karthik Narasimhan
knarasim@student.concordia.ab.ca

Harmanpreet Singh
hsingh35@student.concordia.ab.ca

Jaspreet Ahluwalia
jahluwal@student.concordia.ab.ca

Gadi Harshitha Reddy
greddy@student.concordia.ab.ca

Sai Sushma Peruboina
speruboi@student.concordia.ab.ca

Advisor: Dr. Sergey Butakov
sergey.butakov@concordia.ab.ca

Department of Information Systems Security and Assurance Management
Concordia University Of Edmonton
Edmonton,Canada

***Abstract*— Malware is rapidly spreading on mobile platforms, causing problems for users. Worldwide, 72.72% of users are using android-based smartphones [1]. New malware is created rapidly: obfuscation techniques can evade the signature-based mechanism implemented in current antimalware technology. This paper presents the results of a study that examines how obfuscation techniques affect malicious and benign applications by two widely used malware detection approaches, respectively static and dynamic analysis. The research looked at 5000 samples of malware and benign applications and evaluated the impact of obfuscation on Android applications. Experimental results indicated that up to 73% of the reviewed applications "survived" the obfuscation that increased their chances of evading antivirus detection.**

Keywords*— *android, malware, obfuscation, static analysis, dynamic analysis, android virtual device (AVD), android package kit (APK), malware detection ratio.

I. INTRODUCTION

The number of mobile devices continues to grow exponentially. Android is one of the most popular mobile device platforms, with installation on billions of devices worldwide. As the popularity of smartphones has grown, so have the number of malware applications targeting such devices and alternative Android application repositories that distribute such applications. Consumers often use anti-malware programs to protect their mobile devices, which scan apps for malicious code. However, these products have not always been able to detect malware. Malware creators frequently rely on code obfuscation to prevent detection. Code obfuscation [2] converts code into a more complicated format to decipher, interpret and reverse engineer for humans and computers. Such a modification does not alter the semantics of the code. Code obfuscation may be minor or sophisticated, like bytecode encryption or adding unused code [3]. There are several commercial and open-source obfuscators available on the market [4] [5]. They provide the ability to imply single or multiple code obfuscation strategies to the application code to prevent the reverse engineering of code and protect the intellectual proprietary. However, malware writers leverage the same tools for performing code obfuscation of the malicious code and injecting it inside the benign application to bypass anti-malware tools. The easy accessibility of reverse engineering tools in conjunction with rich bytecode semantics has led to an exponential increase in Android malware.

Consequently, substantial attempts have been made to establish strategies for identifying Android malware. Anti-malware products based on the detection methodology used can be classified based on two broad categories: static and dynamic detection. Static detection analyzes the Android application code through reverse engineering techniques without the Android application (APK) being run. On the other hand, the dynamic detection technique analyzes the application's run time behaviour to detect malicious calls.

This project discusses (1) the effects of single and combined obfuscation techniques on the detection capability of anti-malware products through multiple obfuscation tools, (2) the accuracy of anti-malware product to differentiate malicious and benign apps after transformation, (3) the time impact on the identification of individual items by obfuscated app and (4) the "survival" ratio of malware after subjecting to obfuscation.

II. BACKGROUND

A. Android Architecture

Android supports Java language and enables developers to build an application using available Java libraries. The Android architecture consists of five layers: application, application framework, libraries and Dalvik virtual machines, Android runtime, and Linux kernel [6]. Linux 2.6 is the basis for Android, and the installed applications and device hardware interact with the kernel's help. The Linux kernel handles the functionalities related to storage, power, application and device drivers, network, memory, and process management. The application developer uses the Linux kernel to perform various tasks, ranging from process management to security. The Dalvik virtual machine components' primary function is to execute files with extensions ".dex," developed in Java. The file with "dex" consists of ".jar" and compiled source classes ".class," which is used by the application running on the Android operating system. Application framework consists of services such as activity manager, windows manager, content providers, package manager, resource manager, location manager, and many more and referred to as application programming interface (API) component. While developing an Android application, developers make use of these services to perform the intended activities [7]. The layer that interacts with the end-user is an application, for example, Browser, Settings, Banking application. The security and privacy concerns related to the developed application must be taken care of by the application developer.

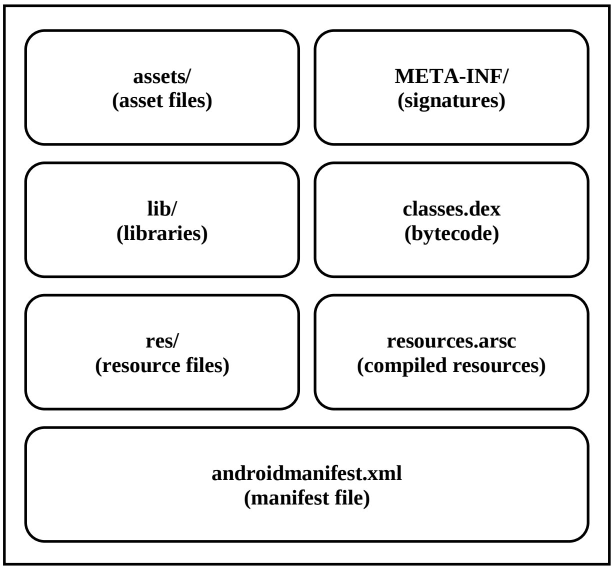

Figure 1 APK file structure [8]

The application running on Android can call or use an element of other installed or running applications. This function can be achieved by the essential components such as activity, services broadcast receivers, and content providers [9]. The subclass for each activity is written, and each activity inherits from the activity class, making it the base class. Services are also considered the main component of any application, and they are running in the background when the application is being used. Whenever an action is requested, a corresponding response is provided with the help of broadcast receivers, and the application may consist of many broadcast receivers to receive and respond to the request.

B. *Android Attack Surface*

An *attack surface* is a primary attribute used to classify if the target is vulnerable to attack based on the risk. An attack vector applies to the way an intruder targets a device. In other words, a vulnerable code can be considered an attack surface. Unlike an attack vector, an attack surface does not depend on the attacker's actions or requires a vulnerability to exist; instead, it describes the places in code where vulnerabilities might be. In general, a target's size is directly proportional to its interaction with other systems. Therefore, a system can be targeted or secured faster if it is focused on risky attack surfaces. Based on the research study, several properties are needed to identify attack surfaces, including attack vectors, memory protection, and access privilege. Also, Remote attack surface is one of the most common attack methodologies used by attackers to gain local or root access to the Android terminal [10].

An attacker can make changes to the permissions specified in the AndroidManifest.xml required by the Android application. APK tempering is a vulnerability that, if exploited, can be mitigated by adding an application code signing mechanism [11]. The Android OS allows developers to sign their applications using a certificate provided by the company that developed the application. After an application is signed, the certificate is used to identify the application, and during communication between the application and the other applications, trust between the two is established. Code signing mechanism is verified while installing the application on the device. Suppose the attacker makes the changes to the existing application. In that case, the attacker will not be able to sign the new build of the modified application with the developer's certificate and restrict installing the modified build on the devices and preventing further attacks focused using the modified build.

C. *Malware Analysis*

Malware analysis is the process of analyzing the malware and studying the components and behaviour of malware. The commonly used malware analysis techniques are static and dynamic malware analysis [12]. Static analysis is a process in which the analysis is done without running the malware, and it is also more secure when compared to the dynamic analysis. In contrast, dynamic analysis is a process of analyzing the malware by running the code, and the process should be done in a more secure environment. Dynamic analysis can be divided into two stages: fundamental analysis and advanced dynamic analysis [13].

a) Static Analysis: Static analysis is the technique that involves viewing the APK file without inspecting the actual instructions. This type of analysis can verify whether the data is malicious, present information about its functionality, and sometimes give information to create some uncomplicated network signature *[14]*. Malware detection is divided into various phases like detection, pre-processing phase, extraction phase, feature phase *[15]*. The feature extraction phase extracts the critical information by parsing the application’s source code to form patterns for classifying the malicious applications.

b) Dynamic Analysis: An application's behaviour can be studied by performing dynamic analysis, also known as behavioural analysis. A few checks typically run during this process, for example, API calls, system calls, network calls, etc. This technique of detection is aimed at evaluating malware in a natural environment by executing the program. Implementing dynamic analysis enables us to identify the dynamic loading of code during run-time and observe the program's behaviour *[12]*. Static analysis techniques cannot calculate code executed during run time. Occasionally, applications can fail to run the malicious code while recording the functions. Instead, an application's source code is run and checked based on the application's actions as soon as it is run. This is useful when the application's source code is obfuscated. It can therefore be used effectively and efficiently in deriving the specific types of behaviour for each malware. However, in addition to signature-based detection on smartphones, antivirus companies think that in-phone analysis is not in the best interest of all parties since scans require limited resources and mobile devices have power and memory limitations.

D. *Obfuscation Strategies*

Malware developers are in the constant race in attempt to avoid detection from antivirus engines. A popular method for achieving this is obfuscation that intends to modify the executable and help the APK evade detection. Obfuscation is also employed by application developers to make it secure from malware authors and to protect the application from being reverse engineered. Research has been done by various authors in this regard and some of them can be reviewed below.

III. Related Works

Rastogi et al. [16] evaluated the efficiency of anti-malware products for detecting malware subjected to trivial and non-trivial obfuscations. The study proved that 10 out of 10 anti-malware products used for research failed to detect the applications that had undergone code obfuscation. The outcomes derived from the research on the obfuscation of malware also showed that obfuscating malware can have a disadvantage which states that the malware will lose its malicious function, causing no damage to its victim's system. Rastogi et al., in their study, found that Anti-malware programs repeatedly failed due to repetitive transformations. Also, Anti-malware tools like VirusTotal lack the capability of developing resilience against the obfuscation method instead of updating its signature database after a malicious variant of the application is detected. Anti-malware tools used in the study took around nine days to detect, analyze, and develop signatures, providing substantial time to damage the Android device. Out of 10 leading anti-malware providers, only 57% of the signatures provided code-level artifacts. The study revealed that 43% of signature identifications were not focusing on code-level artifacts and that component names in the Android manifest were the only way to identify defects. The study also indicates that 90 percent of signatures did not require static bytecode review since much of the information was contained in the classes—dex file of the application with Android runtime code.

In their study, Hammad et al. [17] propose that an anti-malware product's detection capability depends on both the obfuscation methodology used and the tool used for obfuscation. Analysis derived showed that obfuscating the code of an Android application has a significant impact on the top anti-malware product's detection action. The detection rate of top anti-malware products shows a 20% decreased rate when subjected to obfuscation. The combination of multiple security obfuscation techniques does not increase the anti-malware evasion probability over a single transformation. The non-trivial and combined obfuscation detection ratio also remains the same when scanning by top anti-malware products. The results also showed that applications with malware had a significantly less chance of being installed and runnable precisely as in the original form when subjected to obfuscation. Hammad et al. study outcomes prove that applying the correct set of transformations, both trivial or non-trivial, along with commercial obfuscation tools, can have a high anti-malware evasion rate, a more extended survival period less accurate signature detection.

Ajiri et al. [18] looked at the effectiveness of antivirus (AV) engines against Android malware obfuscated. Because anti-malware engines rely on malware analysis for detection purposes, static analyzer detection ratings are evaluated based on their detection effectiveness. His report took each Android malware sample that belonged to 10 different malware families before obfuscation, and their detection ratings were taken. Then, they were compared with obfuscated Android malware by applying three obfuscation techniques, namely string encryption, renaming and control flow individually and their combination. Before obfuscation, Android malware detection ratio values were high and more efficient. Nevertheless, after the implementation of obfuscation techniques individually, their detection ratio decreases significantly, and when the combination of obfuscation techniques was applied, the likelihood of the detection rate was reduced. For example, the research analysis showed that using a combination of obfuscation techniques (Control flow, Renaming, String Encryption), only an average of 23.19% samples out of 50 malware samples (5 samples each under ten families) were detected by around 66 analyzers under VirusTotal. While without obfuscation, the average detection rate was 54.58%. He also mentioned that further step is required for this research study to perform dynamic analysis on obfuscated Android malware to capture their system calls and compare their results with system calls invoked by non-obfuscated Android malware.

In their study, Malik and Khatter [19] proposed that detection of obfuscated malware is insufficient with static malware analysis tools and techniques. System call analysis is a powerful technique for malware that is highly encrypted or obfuscated with other methods. Malicious applications call almost the same system calls with different numbers and perform the same file and network operations during the runtime. Therefore, in their research report, they focused more on the behavioural characteristics of malware. They used a trace tool for system calls extraction and extracted 345 Android malicious APKs that belong to ten Android malware families. In their findings, they mentioned malicious applications initiate more system calls than benign apps, like ptrace() system call is invoked 43561 times by Opfake malicious apps and FakeInstaller applications invoke this system call 39384 times. Malware belongs to different malware families involves a different set of system calls and with different frequencies. They also mentioned system calls that were invoked most frequently like sigproc(), open(), recvfrom(),sendto(), read(), write() etc.

Allix et al. [20] initiated the AndroZoo project that collects millions of Android applications using multiple crawlers and analyzing them for malware. The intention behind the creation of AndroZoo was to provide millions of pre-analyzed applications to the research community for experimental purposes. To access the AndroZoo Repository, permission needs to be acquired from the University of Luxembourg's authorities; when permission is granted, an API Key will be acquired. Then, with the combination of SHA256 values of APK's and the API Key, applications can be successfully downloaded.

The research project presented in this paper aims to study the impact of obfuscation on the malware functionality and detection ratio. Such impacts can be reviewed through the analysis of the installability of the obfuscated software. This is a crucial step because automated malware tools perform "blind" obfuscation that may incapacitate the malware. This research aims to answer the following questions:

1. Can feature extraction prove helpful in identifying APKs that have been subjected to obfuscation?
2. What is the most effective obfuscation method out of the ones being implemented?
3. Which obfuscation method produced the most installable and runnable APKs?
4. Which obfuscation method produced the most non-installable and runnable APKs?

IV. Methodology

The approach for Android malware analysis uses static and dynamic methods along with comprehensible and Obfuscated Android APK files. To accomplish this, a Python script is proposed, which includes all the steps shown below. Furthermore, it involves minimal human involvement and automates all the tasks to generate the CSV/JSON format dataset.

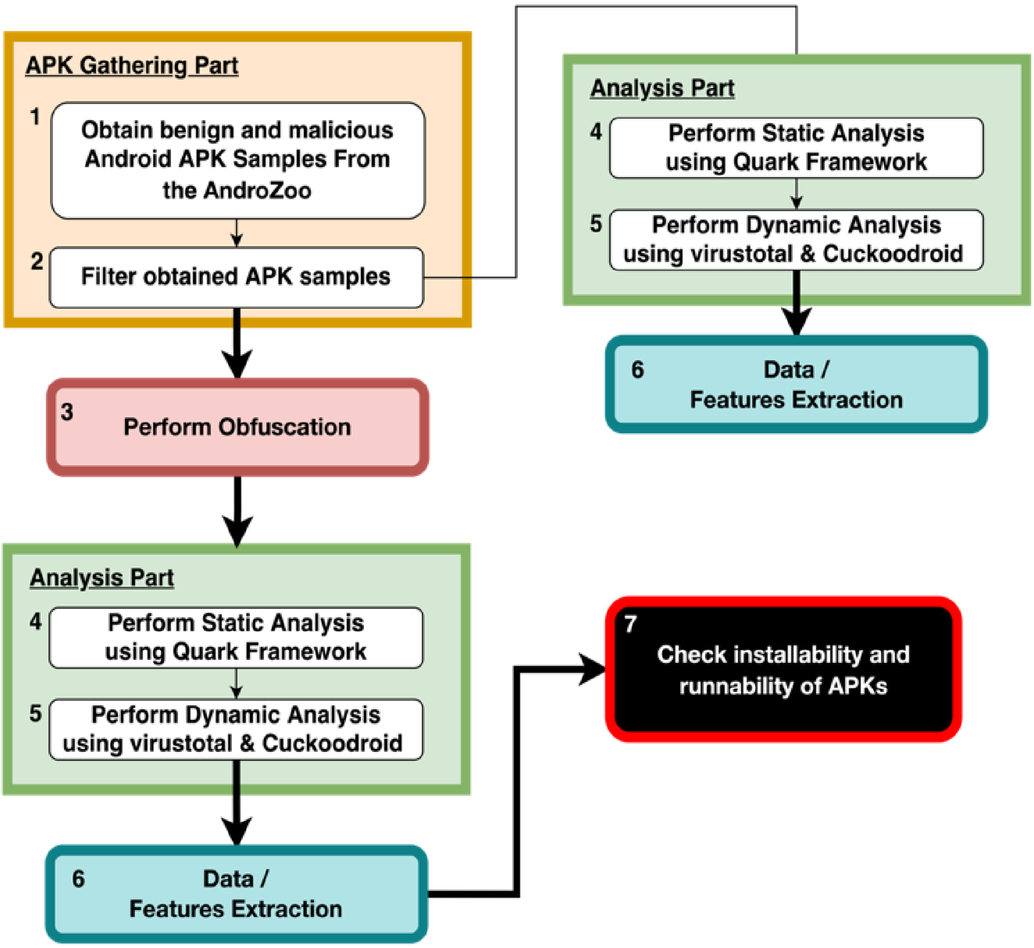

Figure 2 Research Methodology

Step 1. APK Gathering

Android Applications for this experiment were collected from an extensive app database called AndroZoo. AndroZoo is a growing collection of pre-analyzed Android apps that are sourced from several sources, including the official Google Play application market [20].

For this experiment, 5000 APK's from 2013 to 2016 were selected from AndroZoo by filtering them by vt_detection=[0,30+] to obtain higher confidence of malware samples. In addition, different Android application markets such as Google Play Store, slide and anzhi were selected to ensure efficient sampling.

Step 2. Obfuscation

The process of performing transformations on an Android application is called Obfuscation Strategies. These transformations can either be a single or polymorphic transformation. The Android ecosystem has established a categorization of obfuscation techniques into two main groups: trivial and non-trivial [21].

A. Trivial techniques

The simplest obfuscation techniques are the trivial ones. These trivial obfuscation techniques do not change the semantic of the code but can help malware evade specific signatures in anti-malware products. There are four trivial techniques, which include: Align, Re-sign, Rebuild, and Randomize Manifest [21] [22].

NewAlignment	Application code is realigned.
NewSignature	A new custom signature can be used to resign the application.
Rebuild	The application is rebuilt using the new obfuscated parameters.
RandomManifest	The entries in the manifest file are reordered randomly.

Table 1 Trivial Obfuscation Techniques [21] [22]

B. Non-Trivial Techniques

In contrast to straightforward trivial techniques, non-trivial techniques offer a lower detection rate and greater robustness. Resources, including bytecode and other resources (XMLs, asset files, and external libraries), are the targets of non-trivial obfuscation. [22]. There are four subcategories of non-trivial obfuscation techniques: Renaming, Encryption and Code [21].

Renaming: Software should have meaningful names for identifiers such as variables, functions, and so on to enhance readability while maintaining flow. The exact names, however, may expose code functionality. In addition, as the package name uniquely identifies an application, a change to it essentially means that the app is being placed into the

Android ecosystem as a new application. Thus, each identifier is renamed into an obscure and meaningless one, using the renaming technique.

ClassRename	Replace the package name and rename classes
FieldRename	Fields are renamed
MethodRename	Methods are renamed

Table 2 Non-Trivial Obfuscation Techniques – Rename [21] [22]

Encryption: In an APK file, the developer can specify what resources to request at run time. It might be a string or a native library. Code and resources are encrypted in packages and decrypted during the execution phase by the secret keys of obfuscation tools.

AssetEncryption	Asset files are encrypted
ConstStringEncryption	Constant strings in the overall code are encrypted
LibEncryption	Native libraries are encrypted
ResStringEncryption	Resource strings inside the code are encrypted

Table 3 Non-Trivial Obfuscation Techniques – Encryption [21] [22]

Code: Code obfuscation techniques involve modifications to the source code after decompiling that affect instructions inside the classes.dex. Several different techniques have been developed to hide the application's behaviour, each addressing a different aspect of the code [21] [22].

Reflection	In this method, existing code is examined to find invocations of the main application method, while ignoring the Android framework calls. This method can be called using the Reflection APIs if it finds a method invocation that matches a suitable instruction.
AdvancedReflection	Using reflection, dangerous APIs from the Android Framework are invoked.
ArithmeticBranch	Uses junk code insertion technique. A branch instruction is crafted in such a way that the branch is never taken, which results in a piece of junk code composed by arithmetic computations and a branch instruction.
CallIndirection	It adds new methods that invoke the original ones. It modifies the control-flow graph without touching the code semantics.
DebugRemoval	The debug meta-data will be removed using this method.
Goto	The software will insert a goto into the method and a second goto after the first goto at the end of the method so that the control-flow graph will be modified by adding two new nodes.
MethodOverload	This exploits the Java overloading feature to return different methods with the same name, but varying their arguments.
Nop	Random Nop instructions are inserted into every method implementation with this technique.
Reorder	The order of blocks is changed in this technique. An inverted condition and reordered basic blocks are created when a branch instruction is found.

Table 4 Non-Trivial Obfuscation Techniques – Code [21] [22]

Table 5 outlines the 6 different obfuscation strategies implied using Obfuscapk [21] for conducting research

Obfuscation Strategies	Methods
Encryption	AssetEncryption, ConstStringEncryption, LibEncryption, ResStringEncryption
Code	AdvancedReflection, ArithmeticBranch, CallIndirection, DebugRemoval, Goto, MethodOverload, Nop, Reflection, Reorder
Rename	ClassRename, FieldRename, MethodRename
Low	ClassRename, AssetEncryption, AdvancedReflection, MethodOverload, Goto
Medium	ClassRename, FieldRename, ConstStringEncryption, ResStringEncryption, AssetEncryption, AdvancedReflection, MethodOverload, ArithmeticBranch, CallIndirection
High	ClassRename, FieldRename, MethodRename, ConstStringEncryption, ResStringEncryption, AssetEncryption, AssetEncryption, AdvancedReflection, MethodOverload, ArithmeticBranch, CallIndirection, DebugRemoval

Table 5 Obfuscation Strategies

Step 3. Static Analysis

In the static analysis stage, the application is decompiled to obtain four features that are used to classify the application: permissions, native-permissions, intent-priority, and sensitive functions [23]. Android provides permissions [24] as a security feature. Associative functions can be abused if the application wants to execute a specific function without declaring the appropriate permission in Android Manifest.xml. Permissions are used to control applications' functions and to manage the resources of the mobile phone. Android 4.0 includes 153 permissions [24]. Despite this, in a highly free environment, some may utilise this feature to hide the real purpose of applications or embed malicious functions within normal ones for malicious purposes.

ACCESS_BACKGROUND_LOCATION [API level 1]	USE_SIP [Added in API level 9]
ACCESS_COARSE_LOCATION [Added in API level 1]	MODIFY_PHONE_STATE [Added in API level 1]
ACCESS_FINE_LOCATION [Added in API level 1]	WRITE_CALENDAR [Added in API level 1]
CALL_PHONE [Added in API level 1]	INSTALL_PACKAGES [Added in API level 1]
READ_PHONE_STATE [Added in API level 1]	WRITE_CONTACTS [Added in API level 1]
READ_SMS [Added in API level 1]	READ_CALENDAR [Added in API level 1]
RECEIVE_MMS [Added in API level 1]	GET_ACCOUNTS [Added in API level 1]
RECEIVE_SMS [Added in API level 1]	READ CONTACTS [Added in API level 1]
RECEIVE_WAP_PUSH [Added in API level 1]	READ_CALL_LOG [Added in API level 16]
READ_EXTERNAL_STORAGE [Added in API level 16]	WRITE_APN_SETTINGS [Added in API level 1]
ACCESS_MEDIA_LOCATION [Added in API level 29]	RECORD_AUDIO [Added in API level 1]
ACTIVITY_RECOGNITION [Added in API level 29]	CAMERA [Added in API level 1]
ANSWER_PHONE_CALLS [Added in API level 26]	SEND_SMS [Added in API level 1]
BODY_SENSORS [Added in API level 20]	WRITE_CALL_LOG [Added in API level 16]
READ_PHONE_NUMBERS[Added in API level 26]	PROCESS_OUTGOING_CALLS [Added in API level 1 , Deprecated in API level 29]

Table 6 Dangerous permissions [24]

Manifest.xml also defines intent-priority, which identifies the priority of program activities [8]. For example, Application A has a higher intent-priority value than Application B. In that case, related messages will be sent first to A. Most malware raises the intent-priority value to ensure they see information before normal software. Static analysis also examines function calls made by sensitive functions. As part of static analysis, this study analyzes how often sensitive functions are utilized by an application. The table below lists the most common permissions that are necessary to perform static analysis.

Manual verification was also used to verify if any parameters (permissions, activities, services) have changed while comparing the original APK to the obfuscated APK. For instance, using meld software the manifest files of the original APK and the obfuscated APK were compared to find out if any permissions were added or deleted in either of the manifest files. Random APK's were selected from the dataset and the comparison was done between the manifest files of the original APK and the obfuscated version of the same APK. The results are extracted and stored in Microsoft excel for reference.

Step 4. Dynamic Analysis

a) Automatic Dynamic analysis: For dynamic analysis using VirusTotal API, *[25]* original and obfuscated APK's were submitted to VirusTotal and results were retrieved thereafter. The results were fetched and stored in an Excel file in tabular format for ease of analysis. The results were based on execution behavior analyzed by any two of the Android Sandbox namely R2DBox and Droidy used by VirusTotal. The process of submission and retrieving results was done with the help of custom Python scripts to enable large number of sample submission and analysis.

b) Manual dynamic analysis: For manual verification, original and obfuscated APKs were installed and executed in Android Studio to check if the APKs had survived the different obfuscation methods and executed the same as the original ones or not. During the execution of applications, package names under which apps were running, which system calls APKs were calling for the original and obfuscated APKs, which includes system call name, time percentage, usecs/call, frequency, and errors, were recorded *[26]*. System calls help a malware analyst to understand the behaviour of the application. This data extraction was performed with the help of the Strace tool in the adb (Android debugger) shell. Their results were recorded for further analysis. 74 APK samples were randomly selected from the dataset of obfuscated and original APKs.

Step 5. Data Extraction

Data Extraction was embedded as a part of static and dynamic analysis, wherein static analysis quark Framework generated the result in JSON format and Dynamic Analysis excel file were used for logging activity response.

Step 6. Installability

Finally, the original and obfuscated applications were installed on AVDs [27] to check their installability and verify the number of valid applications produced by every obfuscation method. For successful execution and analysis, Anbox and Android Studio were used for loading the applications into them.

C. Design And Implementation Of Script For Methodology Automation

Python scripts are constructed based on a methodology that is customized to the specific requirement.

1) Python Script Flow:

The code continuously works in a loop downloading the APKs (Android Application Package) from the AndroZoo using API calls. Upon successfully downloading the APK file, the function "static analysis" is called. This function uses Quark Framework, which performs the static analysis and generates the report for a particular APK. A report generated by the function is stored in the folder "Report". After a static analysis of the APK has been completed, the APK can then be imported into an analysis function called "dynamic analysis" that uses the Cuckoodroid [28] to analyze and create a report.

Once the APK File has been analyzed both statically and dynamically, it is passed through the Obfuscation function, producing six different obfuscated APK files using six different Obfuscation techniques (Rename, Encryption, Code, Low, Medium, High). To accomplish this modular Python tool, Obfuscapk has been used.

APK files obfuscated by these programs are now again submitted for dynamic and static analysis and reporting purposes. In addition, these files are imported into an emulator to check how they survive after obfuscation. A Python module had been used for the Android bridge driver. Afterward, the user will get a CSV file showing the installed applications and those that did not.

Source code of the script is available at https://github.com/ddeepp109/Android-Malware-Analysis [29]

V. EXPERIMENTAL RESULTS

A. Finding 1: Obfuscation Stratagies

Table 7 shows how different types of obfuscation have a varying effect on the detection ratio. To better understand the impact of every obfuscation strategy on static and dynamic analysis, the original dataset was obfuscated using Obfuscapk with varying levels of obfuscation methods described in Table Y of Section IV. The research outcome showed that the detection rate of VirusTotal on the original dataset is 91%. This detection rate was dropped to 71% on obfuscated apps using Medium, 66% on obfuscated apps using Encryption, and 65% on obfuscated apps using High obfuscation. It was also observed that most of the malware detection was not affected by Low obfuscation.

Obfuscation Techniques	Detection Ratio	Percentage
Encryption	3498/5299	66.03%
Code	3602/5299	67.98%
Rename	3815/5299	72.00%
High	3443/5299	64.99%
Medium	3867/5299	72.99%
Low	4132/5299	77.98%

Table 7 Detection Ration based on Obfuscation Strategies

Another noticeable outcome derived was the impact of trivial and non-trivial obfuscation techniques had almost similar detection rates. A counter intuitive conclusion that can be derived considering an Android APK is an archive with a lot of files and a malicious component can be found almost anywhere, it is not possible to know which of the above-mentioned techniques to be used as a rational, since each technique has different effects on the files contained within.

B. *Finding 2: Impact of Obfuscation on Static analysis*

A random sample of 2000 applications from the benign and malware sets were selected for static analysis. Each APK was decompiled using QUARK [30] to extract five kinds of features: 1. Permission requested. 2. Native API call. 3. Certain combinations of native API. 4. Calling sequence of native API. 5. APIs that handle the same register [30]. Out of the total 2000 APKs subjected for static analysis, Quark detected all the malware APKs in original form. However, the detection ratio reduced to 82% for monomorphic obfuscation techniques while producing the lowest detection ratio High obfuscation of 72%, which had polymorphic obfuscation strategies enabled. Below table 7 defines the detection ratio for varying level of obfuscation.

Obfuscation Method	APKs Tested	APKs Detected	Percentage
Encryption	5299	4371	82.50%
Rename	5299	4398	83%
Code	5299	4191	79.10%
High	5299	3831	72.30%
Medium	5299	3974	75%
Low	5299	4451	84%

Table 8 Detection Ratio based on Static Analysis.

Meld [31] a static analysis tool was used to perform two-way and three-way comparisons of the files. The manifest files were compared to determine the impact of obfuscation on the permissions listed in the original APK. Random APK's were selected from a dataset of 30,000 APKs to check if the permissions were added or deleted from the obfuscated file compared to the original file. In conclusion, all APK's compared have the same permissions in both original and obfuscated manifest files. But in some APK's, although all the permissions are the same in the original and the obfuscated manifest file, the only difference is the number of times each permission is being used in the obfuscated file.

	Permissions	
APK	Original	Obfuscated
xxxxB917	23	21
xxxxD1C8	5	4
xxxx7C07	15	9
xxxx93FF	10	10
xxxxD60C	10	10

Table 9 Comparison of permissions in the manifest file of the original APK with the obfuscated APK.

C. *Finding 3: VirusTotal Dynamic Analysis Findings*

Table 10 shows the results after the samples were uploaded to VirusTotal and after behavioural reports were fetched from two VirusTotal Sandboxes named R2DBox and Droidy. (For detailed results, refer to Appendix B at the end of the report). The analysis done under VirusTotal shows that out of all the obfuscations done, Medium and High levels of obfuscations have shown the most impact on executability of obfuscated samples as 70% were seen showing any behaviour. In contrast, low obfuscations showed more executability as almost 77% of samples produced behavioural results. The single technique obfuscations methods (Code, Rename and Encryption) were shown exhibiting the most execution ratio with 79% of samples producing results. Thus, the ability of obfuscators to produce different variants of a malware sample with fewer detection capabilities and good survival ratios can act as a detrimental tool to bypass specific mechanisms deployed for the detection and protection against suspicious packages.

Samples	Obfuscation	Execution ratio
5299	Encryption	79.39%
5299	Rename	79.36%
5299	Code	79.48%
5299	High	70.48%
5299	Medium	70.04%
5299	Low	77.65%

Table 10 Executability of Obfuscated samples seen under VirusTotal Droidy and R2DBox results

A special feature of Android since API Level 23 is dynamic permission support [32], which allows apps to request, acquire, and revoke permissions as they run. According to this new runtime permission mechanism, static approaches will not be able to detect when abnormal permission requests and grants are made at runtime. In addition, users may revoke dangerous permissions after their apps are installed, which could cause a false alarm.

D. *Finding 4: Manual Dynamic Analysis Findings*

The data extraction from original and obfuscated APKs in this proposed research is derived from permissions and system calls for Android malware analysis. The system calls will be extracted to compare the behaviour of original APKs and obfuscated APKs, as all requests from malicious apps will go through the system call interface [23] before being processed.

The APKs that were not executed at all called only 9 system calls (read, open, close, getpid, ioctl, mprotect, writev, fstat64, clock_gettime) that were used to perform functions like read, open, get process id, file status, clock time, and write operations on the files stored on external storage. The original and obfuscated malicious APKs were using process-related functions like futex, getpid, getuid, gettid, sigprocmask, and prctl. These APKs used sendto() and recvfrom() system calls responsible for sending to and receiving data from remote servers. Other heavily used system calls that were noticed while doing manual dynamic analysis were related to accessing data and perform read-write operations on files stored on external storage and perform memory functions like read, write, open, close, fcntl64, dup, mmap, munmap, stat64, fstat64 etc. These malicious system calls were used most frequently by original and obfuscated malicious APK samples.

Sample	APKs	Original	Obfuscation Methods					
			Code	Encryption	High	Low	Medium	Rename
1	14	86%	86%	86%	86%	86%	86%	86%
2	10	100%	100%	80%	80%	80%	80%	80%
3	10	70%	60%	60%	40%	50%	40%	60%
4	10	60%	60%	60%	40%	40%	40%	60%
5	10	60%	70%	80%	40%	70%	60%	80%
6	10	70%	80%	70%	40%	50%	30%	70%
7	10	50%	60%	70%	30%	50%	50%	60%
AVG	74	70.8%	73.7%	72.3%	58%	60.8%	55.1%	70.8%

Table 11 Executability of APK samples checked with Strace tool in Android Studio.

Table 11 indicates the executability of APK samples manually analyzed with the *Strace* tool in *Android Studio* (for detailed results, refer to Appendix C at the end of the report). It shows that obfuscation methods High, Low and Medium affected the executability of malicious APK samples. The Medium obfuscation method had affected the executability the most and decreased it by 15.7%. The following two obfuscation methods, High and Low, decreased it by 12.8% and 10%, respectively. All other methods showed the percentage of executability almost the same, i.e., near to 70%. Moreover, some obfuscated APKs were successfully executed but affected the working of the Android OS. For instance, while analyzing the APK samples manually, it is noted that 7 % High obfuscated APKs, 5% of Rename obfuscated APKs, 4% of Encryption, and Medium obfuscated APKs froze or slowed down the emulator. Furthermore, the system calls generated by original and obfuscation methods were also recorded to notice if there was any change in their frequency. It is observed that APK samples obfuscated with High and Medium methods generated more system calls in 9% of the APK samples as compared to original and other obfuscation techniques. The encryption method was intermediate because it generated system calls more in 7% of the APKs and least in 9% of the APKs as compared to original and other obfuscation methods. System calls for the remaining majority of the APK samples were almost the same.

E. *Finding 5: Application Installation and Runnability*

For the installability of the applications from Sample 1 that have been obfuscated, they were first installed in *Anbox Application Manager* using the automated script. Out of the 14 applications that have been randomly selected for every obfuscation method, 12 applications were successfully installed every time. The other two applications could not be installed. However, these 12 applications were not runnable on *Anbox*. The applications froze *Anbox* every time they were loaded into the emulator. Additionally, these applications were manually installed in *Android Studio*. All 14 applications were successfully installed and were runnable.

From samples 2-7, the obfuscated applications did not successfully install in the *Anbox* application manager. An error regarding APK signature identification was displayed on the screen for every application. The reason for this can be the higher API level that *Anbox* runs on. This reason was identified because the obfuscated applications were also installed on *Android Studio* with a higher and a lower API level machine. The applications were successfully installed on the virtual Android device with a lower API level of 22 and were not installed on the Android with a higher API level of 24. It is also to be noted that the original applications were successfully installed in *Anbox* and *Android Studio*.

Table 12 shows the information regarding the installability of the applications before and after they are obfuscated. The data took installability information in Anbox, Android Studio and VirusTotal. Results from Anbox and VirusTotal are not considered for summarizing the results because they are automated and sometimes produced no result because of a higher API level than Android Studio that produced results in an apt way. The information given in the table above shows that the obfuscation method “Code” produced the highest number of valid applications post obfuscation with 73.71% valid applications. The “Encryption” method produced 72.28% valid applications. The “Rename” method produced 70.85% valid applications. The “Low” method produced 60.85% valid applications. The “Medium” method produced 55.14% valid applications, and the “High” method produced the lowest number of valid applications by producing just 50.85% valid applications.

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
Sample 1	14 x 7 APK						
Anbox	100%	86%	86%	86%	86%	86%	86%
Android Studio	86%	86%	86%	86%	86%	86%	86%
VirusTotal S1	0%	93%	93%	93%	93%	93%	93%
VirusTotal S2	0%	50%	64%	71%	57%	57%	64%
Sample 2	10 x 7 APK						
Anbox	100%	0%	0%	0%	0%	0%	0%
Android Studio	100%	100%	80%	80%	80%	80%	80%
VirusTotal S1	0%	70%	50%	70%	50%	80%	70%
VirusTotal S2	80%	70%	70%	60%	60%	70%	60%
Sample 3	10 x 7 APK						
Anbox	70%	0%	0%	0%	0%	0%	0%
Android Studio	70%	60%	60%	40%	50%	40%	60%
VirusTotal S1	0%	40%	40%	20%	30%	30%	40%
VirusTotal S2	60%	50%	40%	30%	50%	40%	40%
Sample 4	10 x 7 APK						
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	60%	60%	60%	40%	40%	40%	60%
VirusTotal S1	0%	60%	60%	40%	60%	30%	60%
VirusTotal S2	100%	70%	60%	50%	60%	40%	60%
Sample 5	10 x 7 APK						
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	60%	70%	80%	40%	70%	60%	80%
VirusTotal S1	0%	40%	50%	20%	30%	30%	40%
VirusTotal S2	70%	50%	50%	10%	20%	40%	60%
Sample 6	10 x 7 APK						
Anbox	70%	0%	0%	0%	0%	0%	0%
Android Studio	70%	80%	70%	40%	50%	30%	70%
VirusTotal S1	0%	40%	30%	30%	40%	20%	30%
VirusTotal S2	60%	50%	50%	30%	60%	30%	40%
Sample 7	10 x 7 APK						
Anbox	60%	0%	0%	0%	0%	0%	0%
Android Studio	50%	60%	70%	30%	50%	50%	60%
VirusTotal S1	0%	40%	40%	20%	30%	30%	40%
VirusTotal S2	60%	50%	40%	40%	60%	40%	50%

Table 12 Information regarding Installability of Obfuscated Applications

F. Finding 6 : Comparative Analysis

Figure 3 compares the results obtained from static and dynamic analysis along with installability check. It was observed that applying multiple levels of polymorphic obfuscation bypassed both static and dynamic detection algorithms at the same time caused greater degree of changes in the application semantics resulting in an obsolete malware APK. On the other hand, Trivial and monomorphic obfuscation produced APKs with higher detection ratio but maintained the semantics of the APKs. Overall code obfuscation produced the most optimum results with lower detection ration and higher installation probability.

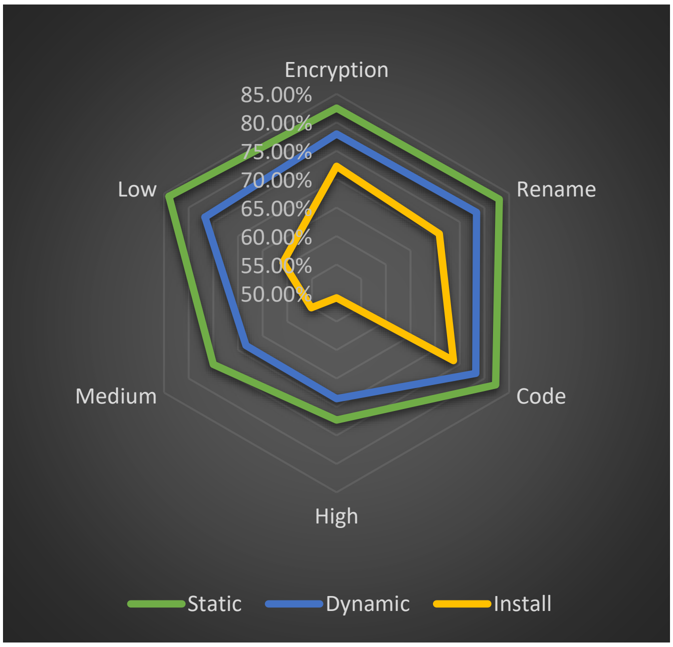

Figure 3 Comparative Analysis of Static Analysis, Dynamic Analysis and Executability of Obfuscated APKs

VI. CONCLUSION

This paper evaluated the effectiveness of static and dynamic analysis against code obfuscation and the survival ratio of malware after varying levels of Obfuscation. From the analysis presented above, it was observed that polymorphic obfuscation techniques had a lower detection ratio as compared to monomorphic obfuscation techniques.

Key findings of the study include the following: (1) regardless of the technique implied, dynamic or static analysis, obfuscation leads to decreased detection ratio of malware code; (2) results obtained from static analysis such as permissions and native API calls produced significantly more information as compared to dynamic analysis (3) in most cases, a trivial transformation, such as modifying the Android manifest file or rebuilding application with a new signature, was effective to bypass detection techniques; (4) despite its relatively weak functionality, dynamic system calls when combined with other features extracted through manual analysis produce effective results increasing the detection ration (5) the APKs' executability was affected by High, Medium and Low obfuscation methods with majority of APKs execution had identical system calls; (6) the applications that were obfuscated with multiple level of obfuscation strategies, High and Medium, to some extent had loss in their application logic and semantics; (7) while monomorphic obfuscation techniques exhibit strong detection resilience, a mixture of obfuscation techniques, polymorphic obfuscation, exhibits an even higher level of detection resilience; and (8) out of all the obfuscation strategies, Code obfuscation proved to be most effective with lower detection ratio and higher installation probability. The results of our study, including the framework developed, are publicly available online.

The experimental setup and results obtained in the paper show that there is a need for an improvement in android security, as both the obfuscation techniques and the tools to manipulate them are readily available to the public. Ease of access to tools and techniques can be leveraged by attackers or script kiddies to execute a successful malware being undetected in case of a targeted attack campaign.

This paper presents data and features generated by the static and dynamic analysis methods, which can be used for future work for a deeper study of how these features can be used to improve the performance of machine learning algorithms for detection and classification purposes.

VII. ACKNOWLEDGMENT

The authors would like to express their gratitude to AndroZoo for providing access to a large APK dataset, VirusTotal (www.virustotal.com), to provide access to Academic API, helping us analyze mass samples and their behavioural summary reports, to Cybera (cloud.cybera.ca) for providing access to their cloud resources.

VIII. REFERENCES

[1] Statcounter Global Stats, "Mobile Operating System Market Share Worldwide," May 2021. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed 12 June 2021].

[2] A. Kovacevic, "What is Code Obfuscation? How to Disguise Your Code to Make it More Secure," 20 NOVEMBER 2020. [Online]. Available: https://www.freecodecamp.org/news/make-your-code-secure-with-obfuscation/. [Accessed 10 May 2021].

[3] C. Collberg, M. GR and H. Andrew, "Sandmark-a tool for software protection research," *IEEE security & privacy 99,* pp. 40-49, 2003.

[4] L. Berzinskas, "Obfuscating Android Apps: Do you know your choices for protection?," 25 Jun 2020. [Online]. Available: https://proandroiddev.com/obfuscation-is-important-do-you-know-your-options-30b3ef396dfe. [Accessed 13 May 2021].

[5] S. Aonzo, G. C. Georgiu, L. Verderame and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for Android apps," *SoftwareX,* vol. 11, no. 100403, 2020.

[6] Google, "Android Architecture Components," 24 February 2021. [Online]. Available: https://developer.android.com/topic/libraries/architecture. [Accessed 13 May 2021].

[7] Android Developers, "Application Fundamentals," 23 February 2021. [Online]. Available: https://developer.android.com/guide/components/fundamentals. [Accessed 14 May 2021].

[8] Google, "App Manifest Overview," 20 April 2021. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed 20 May 2021].

[9] Android Developers, "App Manifest Overview," 20 February 2021. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro#components. [Accessed 20 May 2021].

[10] S. R. Kotipall and M. Imran, "Understanding the app's attack surface," in *Hacking Android,* Packt, 2016.

[11] codemagic, "Android code signing," 15 June 2021. [Online]. Available: https://docs.codemagic.io/code-signing/android-code-signing/. [Accessed 23 May 2021].

[12] O. Or-Meir, N. Nissim, Y. Elovici and L. Rokach, "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey," *ACM Computing Surveys,* pp. 1-34, 2019.

[13] S. Yusirwan, Y. Prayudi and Riadi, "Implementation of malware analysis using static and dynamic analysis method," *International Journal of Computer Applications,* pp. 11-15.

[14] Hacken, "Android Application Malware Analysis," 5 November 2018. [Online]. Available: https://hacken.io/industry-news-and-insights/android-application-malware-analysis/. [Accessed 2021 February 13].

[15] K. Bakour, H. M. Unver and R. Ghanem, "The Android Malware Static Analysis: Techniques, Limitations, and Open Challenges," *3rd International Conference on Computer Science and Engineering (UBMK),* 2018.

[16] V. Rastogi, Y. Chen and J. Xuxian, "Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks," *IEEE Transactions on Information Forensics and Security,* vol. 9, no. 1, pp. 99-108, 2014.

[17] M. Hammad, J. Garcia and S. Malek, "A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products," *Proceedings of the 40th International Conference on Software Engineering,* pp. 421-431, 2018.

[18] V. Ajiri, S. Butakov and P. Zavarsky, "Detection Efficiency of Static Analysers against obfuscated Android Malware," *IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity),* 2020.

[19] S. Malik and K.Khattar, "System Call Analysis of Android Malware Families," *Indian Journal of Science and Technology,* vol. 9, no. 21, 2016.

[20] K. Allix, T. F. Bissyandé, J. Klein and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR),* 2016.

[21] G. C. Georgiu, "Obfuscapk," 17 December 2020. [Online]. Available: https://github.com/ClaudiuGeorgiu/Obfuscapk. [Accessed 25 February 2020].

[22] G. C. Georgiu, .. L. Verder, A. Simone and A. V, "Obfuscapk: An open-source black-box obfuscation tool for Android apps," *SoftwareX*, vol. 11, 2020.

[23] F. Tchakounte and P. Dayang, "System Calls Analysis of Malwares on Android," *Maejo International Journal of Science and Technology*, vol. 2, pp. 669-674, 2013.

[24] Google, "Manifest.permission," 09 June 2021. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission. [Accessed 13 June 2021].

[25] VirusTotal, "VirusTotal API version 3 Overview," [Online]. Available: https://developers.virustotal.com/v3.0/reference#overview. [Accessed 22 March 2021].

[26] H. Yuan, Y. Tang, W. Sun and L. Liu, "A detection method for android application security based on TF-IDF and machine learning," *PLOS ONE*, vol. 15, no. 9, p. e0238694, 2020.

[27] Google Devloper, "Run apps on the Android Emulator," 17 June 2021. [Online]. Available: https://developer.android.com/studio/run/emulator. [Accessed 17 June 2021].

[28] idanr, "CuckooDroid - Automated Android Malware Analysis.," 25 July 2017. [Online]. Available: https://github.com/idanr1986/cuckoo-droid. [Accessed 22 March 2021].

[29] D. Patel, "EVALUATION OF OBFUSCATED ANDROID MALWARE," [Online]. Available: https://github.com/ddeepp109/Android-Malware-Analysis.

[30] quark-engine, "Quark-Engine," [Online]. Available: https://github.com/quark-engine/quark-engine. [Accessed 24 March 2021].

[31] K. Willadsen, "Meld," [Online]. Available: https://meldmerge.org/. [Accessed 12 January 2021].

[32] Google, "Request app permissions," Google, [Online]. Available: https://developer.android.com/training/permissions/requesting. [Accessed 19 February 2020].

Appendix A

Analysis Automation ScriptLink To the Code: https://github.com/ddeepp109/Android-Malware-Analysis/

Shell Script: ***obfuscat.sh***

```
docker run --rm -it -u $(id -u):$(id -g) -v "/home/ubuntu/RM":"/workdir" obfuscapk -p -w /tmp/ -o RandomManifest -o Rebuild -o NewSignature -o NewAlignment $1 -d obfuscatedAPK/$2/$3 APKs/$3
```

In Above command, $1, $2 and $3 represents value, key, and ApkFileName as separate arguments, each of which will be passed through the following Python script.

Python Script: ***flow.py***

```
#importing required packages
import os
import wget

#obfuscation function
def obfuscation(ApkFileName):
    passing_value={
                "Rename":"-o ClassRename -o FieldRename -o MethodRename",
                "Encryption":"-o AssetEncryption -o ConstStringEncryption -o LibEncryption -o ResStringEncryption",
                "Code":"-o AdvancedReflection -o ArithmeticBranch -o CallIndirection -o DebugRemoval -o Goto -o MethodOverload -o Nop -o Reflection -o Reorder",
                "Low":"-o RandomManifest -o ClassRename -o AssetEncryption -o AdvancedReflection -o MethodOverload -o Goto -o Rebuild -o NewSignature -o NewAlignment",
                "Medium":"-o RandomManifest -o ClassRename -o FieldRename -o ConstStringEncryption -o ResStringEncryption -o AssetEncryption -o AdvancedReflection -o MethodOverload -o ArithmeticBranch -o CallIndirection -o Rebuild -o NewSignature -o NewAlignment",
                "High":"-o RandomManifest -o ClassRename -o FieldRename -o MethodRename -o ConstStringEncryption -o ResStringEncryption -o AssetEncryption -o AssetEncryption -o AdvancedReflection -o MethodOverload -o ArithmeticBranch -o CallIndirection -o DebugRemoval -o Rebuild -o NewSignature -o NewAlignment"
                } # commands as a value to perform 6 obfuscation and key is the type of the bfuscation
    for key, value in passing_value.items() :
        obfuscat = "sh ./obfuscat.sh " + '"'+ value + '"' + " " + key + " " + ApkFileName
        os.system(obfuscat)
        static_analysis(ApkFileName,1,key) #calling static analysis function with 2nd argument "1" which define the apkfile is obfuscated
        dynamic_analysis(ApkFileName,1,key) #calling Dynamic analysis function with 2nd argument "1" which define the apkfile is obfuscated

def static_analysis(ApkFileName,n,key="Normal"):
    if n == 0: # value zero for non-obfuscated function
        ApkFilePath = "/home/ubuntu/Android-Malware-Analysis/APKs/" + ApkFileName
        ReportFilePath = "/home/ubuntu/Android-Malware-Analysis/Reports/Static/APK_Report/" + ApkFileName +".json"
        command="quark -a "+ApkFileName+" -s -c -o "+ ReportFilePath
    else:
        ApkFilePath = "/home/ubuntu/Android-Malware-Analysis/obfuscatedAPK/"+ key +"/" + ApkFileName
        ReportFilePath = "/home/ubuntu/Android-Malware-Analysis/Reports/Static/Obfuscapk_Report/"+ key + ApkFileName + ".json"
        command="quark -a " + ApkFilePath + " -s -c -o " + ReportFilePath
    os.system(command)
    print("Static Analysis" + ApkFileName)
```

```
def dynamic_analysis(ApkFileName,n,key="Normal"):
    if n == 0: # value zero for non-obfuscated function
        command="python /home/ubuntu/Android-Malware-Analysis/cuckoo/utils/submit.py /home/ubuntu/Android-Malware-
Analysis/APKs/"+ApkFileName
    if n == 0:
        command="python /home/ubuntu/RM/cuckoo/utils/submit.py /home/ubuntu/RM/APKs/"+ApkFileName
    else:
        command="python /home/ubuntu/Android-Malware-Analysis/cuckoo/utils/submit.py /home/ubuntu/Android-Malware-
Analysis/obfuscatedAPK/"+ key +"/ obfuscated_"+ApkFileName
    os.system("python /home/ubuntu/Android-Malware-Analysis/cuckoo/cuckoo.py --clean")
        command="python /home/ubuntu/RM/cuckoo/utils/submit.py /home/ubuntu/RM/obfuscatedAPK/"+ key +"/
obfuscated_"+ApkFileName
    os.system("python /home/ubuntu/RM/cuckoo/cuckoo.py --clean")
    os.system(command)
    os.system("python /home/ubuntu/Android-Malware-Analysis/cuckoo/cuckoo.py")

#Read the count file to know last processed function
    os.system("python /home/ubuntu/RM/cuckoo/cuckoo.py")

def last_processed_APK():
    file = open("count.txt", "r")
    count = file.readline()
    print (count)
    count = int(count)
    file.close()
    return count-1

#Log the last processed APK
def processed_APK(Number):
    file = open("count.txt", "w")
    file.write(str(Number))
    file.close()

if __name__ == '__main__':
    try:
        print("Downloading APK File....")
        API_key= "fake" #Add Androzoo API Key Here
        count = last_processed_APK()
        file = open("sha256.txt", "r")
        lines = file.readlines()
        print(len(lines))
        LinesInFile = len(lines)
        print (count)
        print (LinesInFile)
        for i in range(count,LinesInFile):        for i in range(count,LinesInFile): # Auto Download APK files from Androzoo
            download1="https://androzoo.uni.lu/api/download?apikey="+API_key+"&sha256="+lines[i]

download1="https://androzoo.uni.lu/api/download?apikey=1fad2754d5ed9728b4f94ea343008c3427830f11a6e55baaa0b951642c
44c6bb&sha256="+lines[i]
            ApkFileName=wget.download(download1)
            print(ApkFileName)
            static_analysis(ApkFileName,0)
            dynamic_analysis(ApkFileName,0)
            print("Static Analysis is Done(Without obfuscation)")
            obfuscation(ApkFileName)
            print("Static Analysis is Done(With obfuscation)")
            print("Back to main")
            processed_APK(i)
        file.close()
    except KeyboardInterrupt:
        print('Hello user you have pressed ctrl-c button.')
        processed_APK(i)
        print("Thank You")

        print("Thank You")
```

Appendix B

Detailed Results of Execution seen under VirusTotal Droidy and R2DBox results –

Batches /Behaviour	Conditions	Samples	Android Version and Year	Successfully Obfuscated			Obfuscation Techniques		
				Low	Medium	High	Code	Rename	Encryption
Batch1	**Total**	1000	Marshmallow 6.0 – 6.0.1 2015 API 19-22	976	976	976	976	976	976
BEHAVIOUR	**Not Observed***			47	46	44	45	48	46
	Observed*			929	930	932	931	928	930
	Executed*			893	895	898	897	893	895
	Not Executed*			36	35	34	34	35	35
	Executable*			96.12%	96.23%	96.35%	96.34%	96.22%	96.23%
Batch2	**Total**	523	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	515	515	515	515	515	515
BEHAVIOUR	**Not Observed ***			128	143	145	117	118	116
	Observed*			387	372	370	398	397	399
	Executed*			276	240	234	294	292	290
	Not Executed*			111	132	136	104	105	109
	Executable*			71.31%	64.51%	63.24%	73.86%	73.55%	72.68%
Batch3	**Total**	480	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	468	468	468	468	468	468
BEHAVIOUR	**Not Observed ***			116	134	133	110	109	111
	Observed*			352	334	335	358	359	357
	Executed*			253	208	211	269	269	264
	Not Executed*			99	126	124	89	90	93
	Executable*			71.87%	62.27%	62.98%	75.13%	74.93%	73.94%
Batch4	**Total**	522	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	516	516	516	516	516	516
BEHAVIOUR	**Not Observed ***			111	124	122	109	107	108
	Observed*			405	392	394	407	409	408
	Executed*			270	219	222	280	281	275
	Not Executed*			135	173	172	127	128	133
	Executable*			66.66%	55.86%	56.34%	68.79%	68.70%	67.40%
Batch5	**Total**	520	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	509	509	509	509	509	509
BEHAVIOUR	**Not Observed ***			137	155	152	134	132	132
	Observed*			372	354	357	375	377	377
	Executed*			281	227	232	284	295	294
	Not Executed*			91	127	125	91	82	83
	Executable*			75.53%	64.12%	64.98%	75.73%	78.24%	77.98%
Batch6	**Total**	520	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	512	512	512	512	512	512
BEHAVIOUR	**Not Observed ***			130	154	151	127	128	128

Batches /Behaviour	Conditions	Samples	Android Version and Year	Successfully Obfuscated			Obfuscation Techniques		
				Low	Medium	High	Code	Rename	Encryption
	Observed*			382	358	361	385	384	384
	Executed*			282	218	228	282	284	283
	Not Executed*			100	140	133	103	100	101
	Executable*			73.82%	60.89%	63.15%	73.24%	73.95%	73.69%
Batch7	Total	480	Jelly Bean 4.1 – 4.3.1 2012 API 16-18	468	468	468	468	468	468
BEHAVIOUR	Not Observed *			108	125	123	107	107	108
	Observed*			360	343	345	361	361	360
	Executed*			270	212	216	274	274	268
	Not Executed*			90	131	129	87	87	92
	Executable*			75.00%	61.80%	62.60%	75.90%	75.90%	74.44%
Batch8	Total	355	Ice Cream Sandwich 4.0 – 4.0.42011 API 16-18	346	346	346	346	346	346
BEHAVIOUR	Not Observed *			35	39	40	31	33	33
	Observed*			311	307	306	315	313	313
	Executed*			259	250	241	273	268	273
	Not Executed*			52	57	65	42	45	40
	Executable*			83.27%	81.43%	78.75%	86.66%	85.62%	87.22%
Batch9	Total	**502**	Ice Cream Sandwich 4.0 – 4.0.42011 API 16-18	**496**	**496**	**496**	**496**	**496**	**496**
BEHAVIOUR	Not Observed *			50	53	56	46	40	44
	Observed*			446	443	440	450	456	452
	Executed*			388	383	378	414	414	418
	Not Executed*			58	60	62	36	42	34
	Executable*			86.99%	86.45%	85.90%	92.00%	90.78%	92.47%
Batch10	Total	**499**	Ice Cream Sandwich 4.0 – 4.0.42011 API 16-18	**493**	**493**	**493**	**493**	**493**	**493**
BEHAVIOUR	Not Observed *			98	110	107	94	98	95
	Observed*			395	383	386	399	395	398
	Executed*			300	256	272	308	299	310
	Not Executed*			95	127	114	91	96	88
	Executable*			75.94%	66.84%	70.46%	77.19%	75.69%	77.88%

**Not Observed = APKS which were uploaded to VirusTotal and whose original as well as Obfuscated did not execute under R2DBox and Droidy*
**Observed = APKS which were uploaded to VirusTotal and whose original or Obfuscated were executed under either R2DBox or Droidy*
**Executed = Obfuscated APKS which were observed as producing results under either R2DBox or Droidy*
**Not Executed = Obfuscated APKS which failed to produce results under either R2DBox or Droidy*
**Executable = Execution rate derived using ((Executed/Observed) * 100)*

Appendix C

Detailed Results of executability results and generated system calls with *Strace* tool:

Batch1:

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	1BE22F2074185914F058387D5ED8B87C02FE222BB4EE1F6125E3517E44B0DB58.apk						
System calls	9	9	9	9	9	9	9
Executed or not	No, but rotate the screen.	No, but rotate the screen.	No	No, but rotate the screen.	No	No, but rotate the screen.	No, but rotate the screen.
2.	3F41BD60C261837BB60D7FFD547ACA3B2F91F9226604D4E1F3AC993DE900E263.apk						
System calls	27	25	26	25	27	27	28
Executed or not	Yes	Yes	Yes	Yes, but slows down the device.	Yes	Yes	Yes
3.	5D7D04CA4F0EFE3F11A0671319349EA464BB18F104F04B56F7CD9A32311D5C53.apk						
System calls	24	25	24	31	24	34	27
Executed or not.	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.	7E6CC7B8906468117BB9C3B7CCACBE37C3D1D826A125FFA3C98D2778BBF6C893.apk						
System calls	26	29	30	30	25	30	28
Executed or not	Yes	Yes	Yes	Yes, but slows down the device.	Yes	Yes	Yes
5.	9E1ABF07E7A30B60FDC2F0ED4181DFBEEB02B82FE65F0B714D28C237B72ECF7D.apk						
System calls	26	24	31	27	24	24	26
Executed or not	Yes	Yes	Yes	Yes, but slows down the device.	Yes	Yes	Yes
6.	7F7F91D206DCFAE4926D4BBFFB985F212145220F19B1572F91451A49CCB9A4E5.apk						
System calls	31	31	27	27	30	31	36
Executed or not	Yes	Yes	Yes	Yes, but slows down the device.	Yes	Yes	Yes
7.	9C19E5F776382E72D87C471198209FC2CBBB173DB7A6D75B461AA31787941628.apk						
System calls	34	34	30	32	35	30	36
Executed or not	Yes	Yes	Yes		Yes	Yes, but slows down the device.	Yes
8.	89BF9D9F272FCFED0B14E05248C2D7B5E930C6FD30235A6DD5A6C2FBD441A39E.apk						
System calls	35	30	29	28	29	29	31
Executed or not	Yes	Yes	Yes, but slows down the device.	Yes	Yes	Yes, but slows down the device.	Yes, but slows down the device.
9.	01612E7ABF107EF81CEC7DEC4DCDAF56311D5DDCD457DC250E341C7B82691425.apk						
System calls	34	31	30	30	30	29	29
Executed or not	Yes	Yes	Yes	Yes	Yes, but slows down the device.	Yes	Yes
10.	9943ACA7EF6A1BCEC0D8746464360E916EE12E8EC8E5F5010DDCC6A04E8DAE33.apk						
System calls	30	30	29	30	30	30	30
Executed	Yes, but slows down the device.	Yes	Yes	Yes	Yes	Yes	Yes, but slows down the device.
11.	DBB7D283A69CFE5913A1DEF31D97110334071765878 9FECF5960CCEE3C74EC92.apk						
System calls	28	31	30	30	30	31	33
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes, but slows down the device.
12.	EB57B7BD48E2D259C05A824F1FB25B8B2473429CFABEB57C6EA4DF998B383B97.apk						
System calls	10	9	9	10	10	9	9
Executed or not	No	No	No	No	No	No	No
13.	15DB45315FA642960E5315889AB5FED54D1A20923A40BCEE89F94D00003620DA.apk						
System calls	30	30	29	29	29	29	30
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
14.	DA76ED42D154EB191642EF2512FDA79D2D0BB9506A8CA46AF9977825F33F4946.apk						
System calls	31	29	28	29	28	32	28
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes, but freeze the device.	Yes

Batch 2

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	00B3AE21A6FEEAE0F616133C7DCBAA0C454990A9B9E4608830F2A3CB66932A5C.apk						
System calls	34	34	24	34	34	33	25
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2.	01D75DB04EFF364547B60159FC0EDA12DFE98011425C7AE333DFB51018EACB91.apk						
System calls	31	13	19	10	9	9	9
Executed or not	Yes	Yes	Yes	No	No	No	No
3.	02CC86E73BE61180FD66AC9625F655217757291BA255ADBE1EFECA2D1F2803EC.apk						
System calls	21	28	22	24	28	27	25
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.	013B524DDD1AC9D2534D287EFD52E84FAC5825AA8E3A241DEF0800C70545E117.apk						
System calls	36	35	16	29	29	30	29
Executed or not	Yes	Yes	Yes, but froze the screen.	Yes	Yes	Yes	Yes
5.	032E0F6D3DA42E9ED1E0FB3189E47465E4730BEFBF5D281BD9DD47CCAF9A8C64.apk						
System calls	34	35	36	36	35	36	36
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
6.	039F8B1CACF7557B855249A4665B653093222928F4A8AC8A7610D3FCC9A9F3E6.apk						
System calls	29	27	29	30	26	29	29
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
7.	0264B77FEE1D56EC642ED76061EF354582E3A689CAEAF5A9B0A4461201FC36AC.apk						
System calls	21	33	25	33	34	34	33
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
8.	0304AF9CFA21048090AA088A283C7797FB7234A8DEF6505B7BD06345AC88FCE8.apk						
System calls	28	30	29	9	9	9	9
Executed or not	Yes	Yes	Yes	No	No	No	No
9.	01733E6D472ECAA38ADC3FA7CD14F7CC720165F2CCD3994C18DCD112A321C44C.apk						
System calls	17	19	20	22	22	22	22
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
10.	02839D0ACF52FE60999C6299D6D170AE2491BF9F9C3848B5124AE8DFC73F4F7E.apk						
System calls	20	22	23	22	22	22	22
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Batch 3

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	04DE1F156F8252312B736E806FF739E9C534E6BF5E2B84450A9E3FB9A2957CF0.apk						
System calls	29	27	28	40	28	31	29
Executed or not	Yes	No	No	Yes	No	No	No
2.	05B2B1F1ED0007BEFBF6B9EBF41DE6DA7480D6388B55E50D2B88B432A688D211.apk						
System calls	24	21	24	9	9	9	24
Executed or not	Yes	Yes	Yes	No	No	No	Yes
3.	053B5A6C0A08FC1BCDEC8FF6D8B1A64D9020B168A31B018842F859E33F797199.apk						
System calls	36	40	15	9	9	9	35
Executed or not	Yes	Yes	Yes	No	No	No	Yes
4.	063FA0B17166CB3BCC5975BAFD528958F7AA1C545931686F7EB51D38128AB80C.apk						
System calls	28	24	35	27	27	16	24
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
5.	04848EB56C4B4301CA97E952398DF561241E844956D7E0D45A5DF015CCDC16C7.apk						
System calls	19	20	19	17	20	19	19
Executed or not	No	No	No	No	No	No	No
6.	06C6BF912683C4E9CAFB083809BCD3A91E8887C757888DDC54D12D07EB2B6CA2.apk						
System calls	33	33	33	33	33	33	33
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
7.	065AC09DB43F5AA56B9650BFA19469F8E2D03508368F15DEE6E9B58C9D491236.apk						
System calls	21	23	22	22	21	22	22
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
8.	067C00C56A412ADE106510E04289E8377412C7D3CB2D6F225C2829A7CBDB02CC.apk						
System calls	10	9	9	9	9	9	9
Executed or not	No	No	No	No	No	No	No
9.	0751273DD8E2E724E30BF837BF66CF6A062A6F56AD8FFDE952EEB77AED088927.apk						
System calls	33	35	36	35	35	36	35
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
10.	0700657141AF5A28329947BF38E3AFEC516616AD7288379738CBF7670EBE8FC9.apk						
System calls	9	0	0	9	9	9	0
Executed or not	No	No	No	No	No	No	No

Batch 4

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	08B8E291F4B1E72D8CA71F27086ABB8F60385E8A22A1073EC7343A5EE990ADAB.apk						
System calls	16	16	15	8	8	8	8
Executed or not	yes	yes	yes	no	no	no	yes
2.	09CFED4D8C08EA1144F1C7A016306D981447FE9771F9C42A74DDA1F0A78D32A8.apk						
System calls	36	19	17	35	34	36	5
Executed or not	yes	yes	yes	yes	yes	yes	yes
3.	086CD34FD0A27889FCE420C9787497BF0ED37677AB73FFFD3E424008C6AAF23C.apk						
System calls	0	0	0	0	0	0	0
Executed or not	no	no	no	no	no	no	no
4.	0805D08434A1593FC469CD2F8F008723DAFF6D74A2348B4483FFB35BD4AAEE3A.apk						
System calls	27	27	31	28	27	30	30
Executed or not	yes	yes	yes	yes	yes	yes	yes
5.	09358B2DAC12C1D218D092D7BA2148F01A984F597499634E50F09C8946FC9DD8.apk						
System calls	26	27	28	26	25	25	6
Executed or not	yes	yes	yes	yes	yes	yes	yes
6.	0A02F83E842C8AAF3F77A229EE9095A1DD72CA040400FA4BBF89B3AC8CCC657C.apk						
System calls	3	3	3	3	3	3	3
Executed or not	no	no	no	no	no	no	no
7.	0A7D26BFB6DCAFBECC94434C1534AC534ED7622BBB7481888A6BF6E164E6240E.apk						
System calls	0	0	0	0	0	0	0
Executed or not	no	no	no	no	no	no	no
8.	0A393841094FFA9A81A1D19F9B21A3AC9550033AD59D182B93E373033579DE41.apk						
System calls	9	9	9	9	9	9	9
Executed or not	no	no	no	no	no	no	no
9.	0AA43A5B1424631DA5C2E6F340E8CC29357940D4784D0CF7E43765CD55E1F8BF.apk						
System calls	26	23	23	9	9	9	23
Executed or not	yes	Yes, but freeze the device	Yes, , but freeze the device	no	no	no	yes
10.	0B0C6B68E6168EBC297D0B70A00F53559AAF76B0DEC600501FB6415A38D59D07.apk						
System calls	36	36	35	36	37	37	37
Executed or not	yes	yes	yes	Yes, , but freeze the screen	yes	yes	yes

Batch 5

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	**0C04A6FF18C152B02B7AF849D56DD0673BFFB0D89244699810E3561B8CA601BC.apk**						
System calls	19	23	28	16	22	24	19
Executed or not	no	yes	yes	no	yes	yes	yes
2.	0C6FD204A8BA0B1558BD09E1AB361AE2E7EB7E9AAC95A56214B13C1E7A51A4F0.apk						
System calls	31	32	32	36	32	33	30
Executed or not	no	no	yes	no	yes	no	yes
3.	0CC5F47D81A0F1D1817F799E53A6E5190DFDD7720E5ECACC2AAF0E81F13B7FC1.apk						
System calls	28	23	23	34	23	32	23
Executed or not	no	no	no	no	no	no	no
4.	0D4C6C17412BB5B5B970C40900C90D9EDBAC0021305B0A3BC2B31B8EA6C73AA4.apk						
System calls	22	24	15	23	24	24	11
Executed or not	yes	yes	yes	no	yes	yes	yes
5.	0DE630901ACF958B473FB97707765AEC63696CF4D7B119C7413AAAAB7306C986.apk						
System calls	27	27	29	31	15	23	29
Executed or not	yes	yes	yes	no	no	no	yes
6.	0E8ADBE37260C4E0DA641332098DA1DAC1B2C5A47E78534F3CDA6FBC380AC209.apk						
System calls	36	36	36	36	36	36	36
Executed or not	yes	yes	yes	yes	yes	yes	yes

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
7.	0E33C4F6286CBCE4DAF26CCD9C5930B5F660D8767718C93A03DF5BB81F8BCC61.apk						
System calls	35	35	36	37	35	36	35
Executed or not	yes	yes	yes	yes	yes	yes	yes
8.	0E0932A010083FD138361B2EFDA34994C12EA073281CE0C47F10EABAE900822D.apk						
System calls	38	38	38	38	38	37	38
Executed or not	yes	yes	yes	yes	yes	yes	Yes, but freeze the screen.
9.	0EA63837421FAA1FD5D32C175CCA28D845DC7157005CF5CDD2F1ABDEC6B6A449.apk						
System calls	35	35	35	36	34	33	35
Executed or not	yes	yes	yes	yes	yes	yes	yes
10.	0F4AF8B26B98FCD5AB6CE28539149472B862698D0F46B09086B2AB35AE20EB06.apk						
System calls	9	9	9	9	9	9	9
Executed or not	no	no	no	no	no	no	no

Batch 6

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	**0F92BB798552F11400E4112ED23A162BCA8EF07A06D74B06549AA598E0ABF646.apk**						
System calls	32	33	32	35	31	31	32
Executed or not	yes	yes	yes	yes	yes	no	yes
2.	10CD7C1D62E743112B849B335454F7667B3EF7CE6ED262DEB43B78537021FAF1.apk						
System calls	33	30	28	22	28	32	30
Executed or not	yes	yes	yes	no	yes	no	yes
3.	101B126E30B2BA02FC0CC85F2CCFCBE4CAF93220CC85421F4B967232EB533E51.apk						
System calls	19	47	19	19	10	19	19
Executed or not	no	yes	no	no	no	no	no
4.	1059B81C0B49A8D66FCF2CF5466A92DCA2238E7819963BE6A4FECDE68B021B50.apk						
System calls	34	34	28	34	8	36	28
Executed or not	yes	yes	yes	no	no	no	yes
5.	1151289173F7ACAE3B77BD225B5FB9902431B4DFAFC80B0416E00236718B7D56.apk						
System calls	0	0	0	0	0	0	0
Executed or not	no	no	no	no	no	no	no
6.	11A8143B89C25D37C9BD1E1FDCDCBB5545E7B520071406F9D4A43A2DD173DEAE.apk						
System calls	0	0	0	0	0	0	0
Executed or not	no	no	no	no	no	no	no
7.	11EC0FCA89A1D2CA996B95F722BC53CB8AFEC24AB06EBDD82C6236D8DCFFB223.apk						
System calls	37	35	38	9	9	9	35
Executed or not	yes	yes	yes	no	no	no	yes
8.	1176CE1052CA7A179B61FB77A0ED586F3B8712FBB9C052B0C0CF0FA82BD10044.apk						
System calls	33	33	37	37	35	36	37
Executed or not	yes	yes	yes	yes	yes	yes	yes
9.	1205777F0BC689207F688A505EC3D6BEB6DB5C075A48575416A74F97FCE1C9FE.apk						
System calls	20	16	18	22	21	22	16
Executed or not	yes	yes	yes	yes	yes	yes	yes
10.	12859368BE0D49D3AAB9E0CC40E52C162FF6497F07BDDC6A629D1F2495B938B6.apk						
System calls	25	24	25	24	25	24	26
Executed or not	yes	yes	yes	yes	yes	yes	yes

Batch 7

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
1.	03ED74BA6A72CD42B9DE8AEC13AE9B0C0395A688C5E8D5F067594A7901C26A43.apk						
System calls	19	24	36	9	9	9	19
Executed or not	Yes	Yes	Yes	No	No	No	Yes
2.	03ED81AF1EC5F1019AA71A86D055D2A6B0E1391794860C3228F1C51A76A41483.apk						
System calls	0	0	0	0	0	0	0

APK	Original	Obfuscation Methods					
		Code	Encryption	High	Low	Medium	Rename
Executed or not	No	No	No	No	No	No	No
3.	03EF2758F8D88E13495B00BB9925B47B496A9D987DD8FD9BB725958C9138825F.apk						
System calls	27	23	21	38	27	39	21
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.	03F36403E08ECE2B89CA7738BF8DBF6ED543E41961CFA5EEF702A125341C0456.apk						
System calls	51	50	50	3	49	50	51
Executed or not	Yes	yes	Yes	Yes	Yes	Yes	Yes
5.	03F50FEEFFB3E002E0CABF8364986E4643B8B12FEE5B668FC7483A350A88B38E.apk						
System calls	28	28	5	24	5	20	28
Executed or not	Yes	Yes	Yes	No	Yes	No	Yes
6.	065AC09DB43F5AA56B9650BFA19469F8E2D03508368F15DEE6E9B58C9D491236.apk						
System calls	20	20	20	19	21	22	20
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
7.	067C00C56A412ADE106510E04289E8377412C7D3CB2D6F225C2829A7CBDB02CC.apk						
System calls	9	9	9	9	9	9	9
Executed or not	No	No	No	No	No	No	No
8.	06C6BF912683C4E9CAFB083809BCD3A91E8887C757888DDC54D12D07EB2B6CA2.apk						
System calls	33	33	33	33	33	33	33
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes
9.	0700657141AF5A28329947BF38E3AFEC516616AD7288379738CBF7670EBE8FC9.apk						
System calls	9	0	0	9	9	9	0
Executed or not	No	No	No	No	No	No	No
10.	0751273DD8E2E724E30BF837BF66CF6A062A6F56AD8FFDE952EEB77AED088927.apk						
System calls	19	19	19	19	19	19	19
Executed or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes