# Auditing Advanced Android Anti-Malware Tools against Sophisticated Evasion Techniques


**by**

**Samrah**

A thesis submitted to the faculty of Information Security,
Military College of Signals, National University of Sciences and Technology, Islamabad,
Pakistan, in partial fulfillment of the requirements for the degree of MS in
Information Security

January, 2020

# CERTIFICATE

Certified that final copy of MS/MPhil thesis written by MS **NS Samrah,** Registration No. **00000206357**, of **Military College of Signals** has been vetted by undersigned, found complete in all respect as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial, fulfillment for award of MS/MPhil degree. It is further certified that necessary amendments as pointed out by GEC members of the student have been also incorporated in the said thesis.

Signature: _____
Name of Supervisor Assoc Prof Dr. Haider Abbas
Date: _____

Signature (HoD): _____
Date: _____

Signature (Dean/Principal): _____
Date: _____

# DECLARATION

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

# DEDICATION

“In the name of Allah, the most Beneficent, the most Merciful"

I dedicate this thesis to my mother, sister, and teachers who supported me each step of the way.

# ACKNOWLEDGMENTS

All praises to Allah for the strengths and His blessing in completing this thesis.

I would like to convey my gratitude to my supervisor, Dr. Haider Abbas, and my co-supervisor Asst. Prof. Waleed Bin Shahid for their supervision and constant support. Their invaluable help of constructive comments and suggestions throughout the experimental and thesis works are major contributions for the success of this research. Also, I would thank my committee members; Dr. Syed Amer Ahsan Gillani and Asst. Prof. Mian Muhammad Waseem Iqbal for their support and knowledge regarding this topic.

Last, but not the least, I am highly thankful to my parents. They have always stood by my dreams and aspirations and have been a great source of inspiration for me. I would like to thank them for all their care, love and support through my times of stress and excitement.

# ABSTRACT

Mobile malware threats have become a real concern. Malware authors are coming up with smarter ways to build applications that can easily compromise confidentiality, integrity and availability of the user's data and perform other illicit activities like identity theft, financial gains, cyber terrorism etc. The aim of this research work is to audit known antimalware solutions for their efficacy against sophisticated malware evasion techniques. Evaluation of the state-of-the-art commercial mobile anti-malware products for Android is necessary to test how resistant they are against various evasion techniques (even with known malware). Such an evaluation is important for not only measuring the available defense against mobile malware threats but also proposing effective, next-generation solutions. This research work highlights and compares, in detail, various sophisticated techniques employed by the hackers to evade malware detection, along with pros and cons of each technique. It also presents comparison of existing anti-malware tools and their efficacy against the discussed evasion techniques. Finally, using sophisticated anti-malware evasion technique developed for Android Operating System (OS) that uses exhaustive obfuscation to deceive static and dynamic detection respectively to audit known anti-malware solutions and making them more resilient and powerful.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background

Mobile devices have grown into an essential part of day-to-day life. They offer a lot of handy utilities such as the facility to read and write e-mails, surf the Internet, show adjoining amenities, video conferencing, and voice recognition, to name but a few. The rise in the spread and use of mobile phones has played a pivotal role in unveiling this consummate paradigm shift in the way humans communicate globally. Since users' digital life resides on these smart phones, the criticality and sensitivity of that data ultimately becomes dependent on the Operating System these phones run on. Android [1], owned by Google, is among the most popular and the most widely used platforms [2] deployed on smartphones with more than 2 billion active devices [3]. In several circumstances, the usage of word Android is quite precise. Though refers to a humanoid robot, Android has garnered meanings beyond that in the last decade. A company, a development community, an open source project are all the terms related to Android more than just an operating system. In a nutshell, an all-inclusive ecosystem equals a standard mobile operating system which we call Android [2].

### 1.1.1 Android Threat Landscape

The vast usage of Android, along with its open source nature [2] has made it a lucrative option for developers with malicious intent to write and spread malicious code. Consequently, this malicious code is then used to compromise the confidentiality, integrity and availability of user data. In various regards, mobile devices offer pronounced security and confidentiality disquiets to users than conventional PCs [4]. Such as, numerous sensors integrated within the Android device could leak extremely sensitive and significant information from user's location, movements, and other physical conducts, to audio and video recordings, and capturing pictures. Besides, users progressively enclose certification credentials into their gadgets, and employing on-platform micropayment machineries such as Near Field Communication (NFC) [5].

#### 1.1.1.1 Third Party Applications

One chief cause of confidentiality and security glitches is the capacity to feature third-party applications, not only from open markets available online but also by other channels. Two prototypes of smart devices based on user's access to these markets [6] exist at present. In the open-market prototype, applications are installed from online unofficial sources, whereas the supposedly walled-garden market model confines the market from which users can install applications such as Google Play Store for Android. Many market operators perform a review procedure over uploaded apps, which apparently also encompasses some practice of security analysis to identify whether the app contains malicious program. A noteworthy section of users count on other sources to have access for free apps that cost money in authorized markets. Access to such informal and/or illegal markets have paved ways for the malware to have easy access to mobile devices and perform their malicious intent. This is particularly true for the well-known apps altered (repackaged) and updated with malicious code imbedded in them [7]. Figure 1 illustrates a list of applications being exploited along with the attack percentage for these applications in the specified period. Android OS stands third in this list [8].

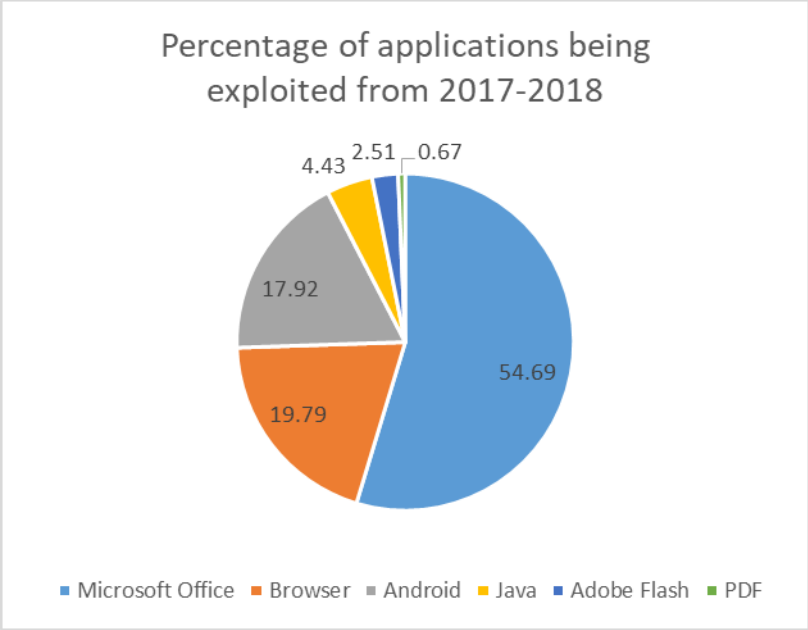



**Figure 1: Percentage of applications being exploited from November 2017 to October 2018**

#### 1.1.1.2 Android Malware Statistics

According to Symantec Internet Security Threat report (ISTR) [9], number of new mobile malware variants grew by 54 percent in 2017 and the mobile malware families grew by 12

percent as compared to 2016. An average of 23,795 malicious mobile applications are estimated to be blocked on mobile devices each day [10].

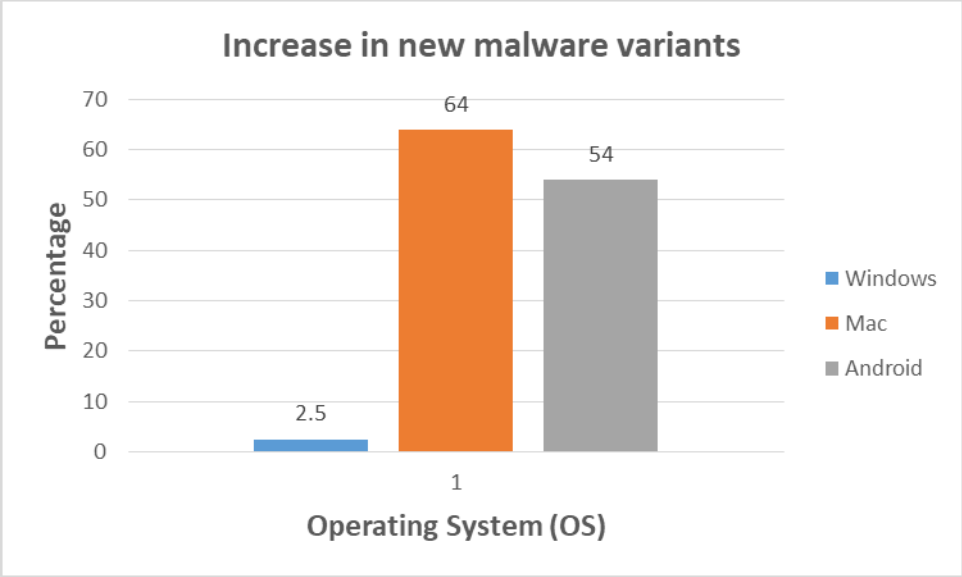



**Figure 2: Percentage Increase in malware variants (2016-2017)**

Moreover, Android rolls its Operating System (OS) upgradation on yearly basis to enhance user experience, security, optimization and device performance. This upgrade is specific to Mobile device vendor, phone model and users' geographic location. Hence, many Android smartphones keep on running the older OS versions. ISTR reports [9] that only 20 percent Android devices are running the newest version as compared to iOS devices where approximately 77.3 percent devices are running the latest version as illustrated in Table 1. This alarming situation, thereby, makes it easier for attackers to compromise devices using the older Android versions.

**Table 1: Percentage of Devices Running Newest Version of Operating System (OS)**

| OS Version | Android | iOS |
|---|---|---|
| **Newest Major** | 20% | 77.3% |
| **Newest Minor** | 2.3% | 26.5% |

### 1.1.2 Android Malware Evasion Techniques

Malware authors deploy several evasion techniques in order to avoid detection by the antivirus programs and other security solutions and in this campaign, new stock of malware variants emerge that are evasive in nature. These devious malwares tend to stay hidden while successfully carrying out their desired illicit action. Some existing malware evasion

techniques include: packing [11], obfuscation [12], steganography [13], code reuse attacks [12] etc. Therefore, studying these canny antivirus evasion and bypassing techniques is of utmost significance.

## 1.2 Motivation and Problem Statement

Malware nowadays have become more advanced, malign and difficult to catch. Malware analysis and detection have appear to be in a competing position, where malware authors aim to hide their malicious intent from security analysts. In this campaign, new stock of malwares emerge which can be defined as evasive malware. Malware authors deploy several evasion techniques in order to avoid detection by Antivirus Programs and other security solutions. Some existing malware evasion techniques include packing, obfuscation, fragmentation, code reuse attacks, application specific violations, protocol violations, traffic insertion at Intrusion Detection System (IDS) and denial of service. Studying how malwares are evading and bypassing security solutions is of utmost significance these days. Every evasion technique has certain limitations and malware analysts are coming up with new detection mechanisms to detect evasion and impede the efforts of malware authors.

In order to evaluate the efficacy of the current state-of-the-art AMTs, we need to develop sophisticated evasive malware in order to audit these AMTs. This will address flaws in the detection mechanisms of these tools and hence improve their detection capability. Also, no standard method/framework, to evaluate the detection competence of these AMTs, exists and existing malware repositories such as Genome [14] and Drebin [15] lack new malware variants. Absence of automation for updating malware repositories is yet another reason for creating new malware variants using a system that will automatically update the malware repository.

This research is focused on auditing Android antimalware solutions against static analysis using a hybrid evasion technique. The technique is amalgamate of various obfuscation modules implemented in an iterative manner. The technique effectively evades static analysis in iterative steps.

## 1.3 Project Description

### 1.3.1 Objective

The aim for malware authors is to evade detection from security analysts. Being security analysts, we need to stay ahead of malware authors and thus thwart their motives. Hence by understanding and developing offensive security methods, we can develop enhanced security mechanism for malware detection and thus evade the malware evasion which can pose serious threats by staying undetected and executing their malicious intent.

The main objectives of thesis are: -

- To conduct a critical analysis of existing evasion techniques for Android malware and evaluation of android anti-malware tools against existing evasion techniques.
- To generate new, advanced and hybrid evasion technique for the class of Android malwares with an aim to audit the new advanced anti-malware tools.

### 1.3.2 Approach

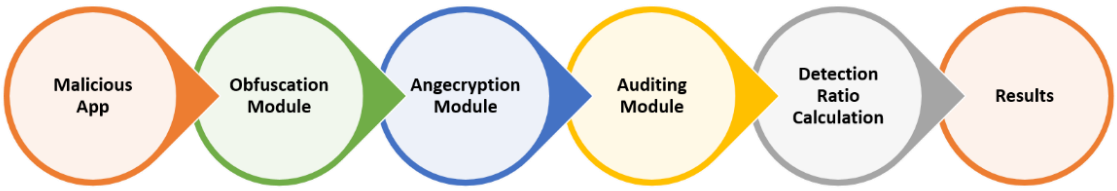



**Figure 3: Basic Approach for Auditing Android AMTs**

### 1.3.3 Academic Objectives

We constantly need to update ourselves regarding new and persistent threats from adversary. When we are well acquainted with new incoming evasive malware threats, we can develop

better detection mechanisms, more transparent ones and stay better guarded against such adversarial motives.

### 1.3.4 Scope of the Project

The focus of the thesis will be to conduct a critical analysis of existing Android malware evasion techniques and anti-malware tools capabilities against these evasive malwares. A technique comprising of new, advanced and hybrid Android malware evasion will be developed. The evasive malware will be deployed on Android devices with advanced anti-malware tools installed followed by an evaluation procedure. This will, thus, conform the anti-malware tool's efficiency against advanced anti-detection techniques.

### 1.3.5 Areas of Application/Advantages

This era is of cyber warfare, one constantly needs to be updated about new threats and their countermeasures. When we know evolving offensive security paradigms, we can develop better defensive mechanisms. Areas of application could be commercial, military and defense.

This will aid in hardening the system security, devising enhanced security mechanism against new class of evasive malwares by determining the possible ways for making malwares as evasive as possible and to innovate/upgrade the existing security mechanisms.

## 1.4 Thesis Organization

The thesis is structured as follows:

- Chapter 2 focusses on Android fundamentals with a focus on Android system architecture, Android application taxonomy and Android security structure and security vulnerabilities.
- Chapter 3 outlines different kinds of malware and malware detection techniques for mobile devices.
- Chapter 4 describes in details several malware evasion techniques employed for Android malwares and summarizes the literature review in regard of malware evasion techniques and auditing of the Android Antimalware Tools (AMTs).
- Chapter 5 explains the proposed framework for auditing Android (AMTs) using sophisticated evasion technique.
- Chapter 6 lists down the prerequisites for the implementation of the proposed framework, software and hardware requirements, malware dataset used, and AMTs employed.
- Chapter 7 pronounces the details of practical implementation of the proposed evasion framework.
- Chapter 8 details the auditing results on several AMTs and their detection efficacy. Also comparison with other works is also presented.
- Chapter 9 concludes the work and presents future directions for strengthening Android Antimalware engines.

# Chapter 2

# Android Fundamentals

## 2.1 Android System Architecture

The term “Java on Linux” coined for Android system architecture is a bit of a loose term to explain the complexity and architecture of the platform. Android's foundation is the Linux kernel. Several add-ons and variations were performed to Linux kernel resulting in certain security implications. Android’s architecture consists of five main layers of components, namely framework, applications, user-space native code, the Dalvik Virtual Machine (Dalvik VM), and the Linux kernel. Fig. 4 illustrates the basic Android Architecture [11]. Linux Kernel resides at the base of the software stack and includes drivers for audio, IPC, Wi-Fi and USB, memory and process management, network stack etc. Hardware Abstraction Layer (HAL) lies on top of Linux kernel and acts as an abstraction layer between hardware and software. The succeeding level of the structure comprises the libraries, a set of directives for handling different types of data. These native libraries include a set of C/C++ libraries comprising of the core libraries such as the System C library, media libraries, and LibWebCore (for a Web browser engine). For instance, the media framework library handles media entities like pictures, video, and audio. A set of core Java libraries constitutes the Android Runtime [16].

Exhausting the Java programming, Android applications are developed. The Application Framework necessary for and accessible to the Android developers includes modules that accomplish the device’s rudimentary jobs like telephone, navigation and resource allocation. The Application stack aids user’s interaction with the device.

Android applications let developers have access to device’s underlying hardware such as Bluetooth, camera, sensors etc. to encompass and improve a device’s function without modifying the lower levels. In sequence, developers facilitate themselves using the Android Framework which provides a rich source of API having right to use all of the innumerable

services an Android device offers. In short, Android Framework acts a glue between the Dalvik VM and apps.

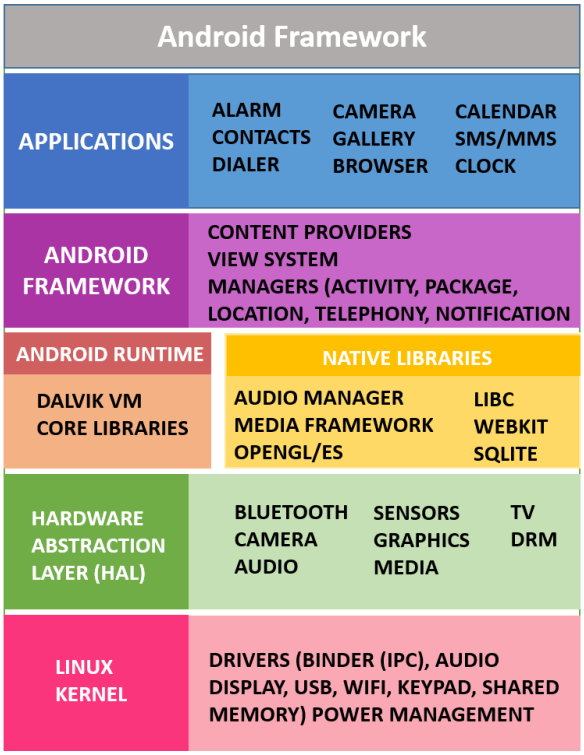



**Figure 4: Android Software Stack**

For instance, it includes allowing developers to perform trivial tasks such as passing messages between application counterparts, handling elements constituting user interface (UI) and having access to shared data stores. Java forms the basis for both the Android Framework and Android applications and these run within Dalvik VM.

Dalvik VM has been designed to make available a resourceful abstraction layer to the core OS. It is used to interpret Dalvik Executable (DEX) using a registered-based VM. In sequence, Dalvik VM rests on the utilities which are provision of several supporting native code libraries.

System services such as networking services and libraries such as OpenSSL, Webkit etc. are the constituents of user-space native code. Few services and libraries have functions to communicate with kernel-level drivers and services while some other aid inherent tasks for managed code. Wi-Fi, camera access and network device access etc. are the add-ons

facilitated by the drivers at Kernel-level. Among these kernel-level drivers, Binder driver responsible for implementing inter-process communication is most important.

## 2.2 Android Application Taxonomy

Mobile applications can be classified as user-installed and pre-installed [2].

- Applications such as Google, Google Play Store and applications installed by mobile carrier such as email, clock, camera, gallery, dialer, contacts etc. come in the category of preinstalled applications as these are already present on the phone even before a user buys it. These applications' packages are located in the /system/app directory and most of these have elevated privileges and cannot be uninstalled by the normal uninstall option.

- Second category is of user-installed applications or third party applications. This class includes applications installed by the user themselves either through an official app market such as Google Play Store for Android OS and App Store for IPhone or through some unofficial source. Such apps reside along with updates for the pre-installed app, in the /data/app directory.

Moreover, public key cryptography is used for signing the Android applications. For signing the pre-installed applications, a special platform key is used which provides these applications with system user privileges. On the other hand, third applications' signing is performed with developers' key. This signing of apps inhibits unapproved updates to the apps.

### 2.2.1 Major Application Components

Android applications are composed of several components. In this portion, we mention few important among them such as AndroidManifest, activities, services, broadcast receivers, intents and content providers.

#### 2.2.1.1 AndroidManifest.xml [2]

An AndroiManifest.xml file is a requisite for all Android applications. This file gives a handful insight into the application such as all necessary permissions declared, package name

unique for each application and its version, services, activities, information about instrumentation, shared User ID (UID), UI info etc. Moreover, the info on external libraries bundled with and consumed by the application and install location favored is also listed in this file.

#### 2.2.1.2 Intents [2]

Intents, a significant fragment of inter-app communication, encompasses info about tasks that are required to be executed, target constituents on which to take action and added flags or further auxiliary info substantial for the recipient. Trivial tasks from installing and uninstalling applications, to notifications about incoming SMS messages, from launching a browser to tapping a link, everything include intents being distributed round a system.

#### 2.2.1.3 Activities [2]

A screen with a user interface is called an activity which is a fundamental component of an Android application with a GUI. Applications may comprise of a number of activities and are put on show in a particular order with each activity with an autonomous launch control, even by a different app if permitted.

#### 2.2.1.4 Broadcast Receivers [2]

A component sensitive to system-wide events, called broadcasts and responds to them is a broadcast receiver. Broadcasts can be initiated by either the system such as announcing changes in network connectivity, or by a user application such as announcing completion of an ongoing background data update.

#### 2.2.1.5 Services [2]

An element of an Android application without any user interface, executing in the background is a service. Time consuming actions such as a file downloading, playing music without halting the user interactions are usually executed by services. Services, dissimilar to system services which are part of the OS and constantly executing, application services can be initiated and halted when required. These can also offer some functionality to other apps and declare remote interface using AIDL.

#### 2.2.1.6 Content Providers [2]

Provision of an interface to app data is the responsibility of content providers. These content providers are either held at some database or stored in files. Employing IPC, content providers can be accessed for sharing app's data with other apps. Controlled accession to an app's data is also a provision of content providers, a utility that ensures the sharing of only a subset of app's data.

## 2.3 Security Model [1]

Android security model is based on that of Linux kernel. Provision of isolated user resources is a feature of Linux security which ensures without explicitly granting permission, one user's resources cannot be accessed by others. Moreover, each process executes with a unique user (UID) and group ID (GID) of the initiator who started the process.

In order to get an insight on the working of AMTs, essential components of the Android Security Model are briefly described in this subsection. The Android Security model is based on application sandboxing. Android achieves application sandboxing by means of Linux User IDs (UIDs) [16]. Every application that runs on Android is assigned a set of attributes such as unique UID, application runtime and application framework. These attributes help the application execute within Dalvik VM [16] which acts as sandbox and isolates the application from other applications. Sandboxed applications communicate with each other and the system according to the Android's Permission Model which uses intent filters to control the permissions explicitly declared in AndroidManifest.xml file or set-group-ID (SUID and SGID)[16] bits are set on the corresponding executable file. Some of the security features of Android are discussed below:

### 2.3.1 Application Sandboxing [1]

A unique User ID (UID) is automatically assigned to each application at install time under which the application executes in a dedicated process. This provides application isolation at the process level. Moreover, under this UID an exclusive data directory with the permission to read and writes to is assigned to each application which provides application sandboxing at the file level. Despite the execution environment being native or virtual, application sandboxing is implemented on all applications.

### 2.3.2 Permissions [1]

Due to application sandboxing in Android, each application have access to their specific files and other resources residing on the device. This limits the application's functionality, hence, to provide applications with more resourceful functions, surplus, controlled access rights are provided. We call these access rights permissions, which control access to hardware devices such as sensors, services such as internet connection, data, or other OS related services. By enlisting permissions in their AndroidManifest.xml file, applications define their set of requested permissions. Android versions running higher than API Level 22, no prior permissions are required at install time, rather permissions are requested at runtime [17].

### 2.3.3 IPC [1]

IPC refers to inter-process communication which is implemented using a set of user space libraries and kernel-level drivers. Forging of the User ID (UID) and Process ID (PID) is inhibited by the Binder kernel driver. This Binder driver also provides several services which provide dynamic access control to several sensitive APIs exposed by IPC.

### 2.3.4 Code Signing and Platform Keys [1]

All Android applications inclusive of system apps are required to be signed by their developer. Due to their dependence on Java and JAR package formats [18], Android applications are signed using signing method based on JAR signing. Using the same origin policy, Android employs the APK signature to ensure updates for an app are from the same author to avoid forging or updates from malicious sources.

### 2.3.5 Security Enhanced Linux (SELinux) [1]

Implemented as Mandatory Access Control (MAC) for Linux, an altered SELinux version from Security Enhancement for Android (SEAndroid) project [19] is integrated in Android. This modified version of SELinux provides features specific to Android such as isolation of core system daemons and definition of distinct access policies for each security domain.

### 2.3.6 System Updates

Updates to the Android devices can be performed in two ways: either over-the-air (OTA) or via establishing connection with a PC through Android Debug Bridge (ADB) or some other application provided by the vendor and pushing updates to the device. Components such as

bootloader, baseband firmware and several other counterparts may also need updating in addition to system services. This is done using recovery mode which employs an exclusive, nominal OS with root access to device's hardware components.

### 2.3.7 Verified Boot [1]

Provision of verified boot in Android version 6.0 and later is ensured via the device-mapper-verity (dm-verity) [20] which is a kernel-level feature. Authenticity and integrity of each upcoming stage before its execution is a feature of verified boot. A strict enforcement of verified boot in Android 7.0 and later ensures the failure of a comprised device's boot. This ensures the integrity of the booting device.

### 2.3.8 File System Permission [2]

This feature ensures that files generated and owned by one application can't be read or altered by some other application until that application assigns permission to have access to its file system by other applications.

### 2.3.9 Rooting of Devices [2]

Certain applications and kernel execute with the exclusive permissions in Android. These root permissions can provide an application with the right to alter OS, kernel or other Android applications and can have access to otherwise inaccessible resources.

### 2.3.10 Device Administration

Device Administration utilities at the system level are a feature of Android 2.2 and later versions.

### 2.3.11 File System Encryption

Starting from Android 3.3, encryption of user files at the kernel level is provided. From Android 5.0, full disk encryption employing single key is performed, which either could be the password for user's device or generated from it. However in Android 7.0 and later versions, distinct and unique keys are employed for encrypting different files which ensures better security compared to single key encryption method.

## 2.4 Security Vulnerabilities

Android has outnumbered Windows platform in terms of its popularity and usage. Owing to the huge amount of user's data, its sensitive and critical nature, a greater threat to its security and privacy exists. In order to fulfill the malicious intent, security vulnerabilities found in the Android Platform are exploited leading to user data theft, encrypt devices, remote code execution etc.

A common identifier for defining the vulnerability, known as Common Vulnerability Exposure (CVE) ID are used by all the vulnerability databases [22]. A Common Vulnerability Scoring System (CVSS) is assigned to determine the impact level of the vulnerability. The vulnerability impact level may be categorized as Critical, High, Moderate, Low and No Security Impact (NSI). The monthly Android Security Bulletin maintains database of evolving Android based vulnerabilities and respective security remediation. The vulnerabilities have been divided into four main categories [16].

Table 1 below lists some of the severe security vulnerabilities, recently found in Android's 'Framework.

**Table 2: CVE Android Security Bulletin, Year 2019**

| Month | CVE | References | Type |
|---|---|---|---|
| July 2019 | CVE-2019-2104 | A-131356202 | RCE |
| June 2019 | CVE-2019-2090 | A-128599183 | EOP |
| Apr 2019 | CVE-2019-2026 | A-120866126 | RCE |
| Mar 2019 | CVE-2019-2004 | A-115739809 | ID |

### 2.4.1 Elevation of Privilege (EoP) [17]

Attacker gains access to protected services/ resources by exploiting vulnerabilities in OS or applications. An exclusive access to a service or a resource usually inaccessible or secured from conventional applications. The malignant application thus bypasses the permissions and gains access to otherwise unavailable and critical data of the users and the system.

### 2.4.2 Remote Code Execution (RCE) [17]

It allows an attacker to remotely execute commands or code of his choice on the target device.

### 2.4.3 Denial of Service (DoS) [17]

Attacker exploits OS/ application to make authorized resources/ services unavailable to legitimate users.

### 2.4.4 Information Disclosure (ID)[23]

Attacker gains valuable information regarding system or user thereby causing privacy issues and information leakage.

# Chapter 3

# Malware, Detection and Analysis

## 3.1 Mobile Malwares

Mobile malwares include Trojans, Backdoors, Ransomware, Botnets and Spyware. Nearly one-third fraction of smart phones has a moderate to an excessive risk of data theft. Moreover, the percentage of Android devices infected with malware is nearly double relative to iOS devices. Some of the most important mobile malwares are listed below:

### 3.1.1 Trojans

A software that executes malicious acts in the background though it appears benign on the surface is called a Trojan [24]. Trojans hack a system by putting the security of the system at stake. Examples include FakeNetflix [25], an Android Trojan responsible for pocketing users' Netflix account credentials and KeyRaider[26], an iOS Trojan used for stealing Apple IDs and passwords.

### 3.1.2 Backdoors

Backdoors takes advantage of root privileges to bypass antiviruses. One popular Android backdoor is Rage against the cage (RATC) which completely hijacks the device and performs exploits [27]. After gaining full control of the device and root access to system's resources, malware can perform tasks capable of even installing applications in the backend not leaving any detectable traces of its action. Similar to RATC, Xagent[28] is an iOS Trojan capable of opening backdoors on iOS devices and information theft from these [29].

### 3.1.3 Ransomware

Users are inhibited from accessing their data by encrypting this data or by locking the device using ransomwares and can only access this data upon paying a ransom. FakeDefender.B [30] is a ransomware, disguised as Avast antivirus, that locks the user's device till ransom is payed. Similarly an iOS ransomware appeared in 2017 that feats on a bug found in Safari pop-ups [31].

### 3.1.4 Botnets

Using a compromised devices, this malware helps attacker hijack the device and then further infect other devices. Web robots, a term used for an affected device, infect all the devices in a network and form a botnet. One such example of Android botnet is Genimi [32].

### 3.1.5 Spyware

As the name refers, spyware is a software used for spying. While executing at the backend without being noticed, it gathers valuable information while also granting remote access in some scenarios. From listing keystrokes, stealing credentials, to collecting browsing history and intercepting communication, a spyware can collect valuable information and send to the attacker. Examples include Nickspy [33], GPSSpy [34] which are spywares for Android whereas Passrobber[29] is an iOS spyware.

## 3.2 Malware Propagation Techniques

To abate malware attacks, we need to be well-equipped with the knowledge of their propagation mechanism. According to [35], malware propagation can be categorized into techniques listed below:

### 3.2.1 Repackaging

By disassembling and then repackaging widely used Android applications while embedding malignant sections into these, and then dispersing these repackaged malware variants as updates to the original app both in the official application hub and less guarded open markets, one can easily propagate malwares. Using tools like apktool, dex2jar and some open-source RATs, one can easily distribute their malware and users often buy this idea assuming updates to the already installed application. According to TrendMicro, more than 70% of the top 50 free apps uploaded to Google Play are repackaged versions [36].

### 3.2.2 Drive by Download

Inadvertent download of a malware at the backend when a user browses a website embedded with malignant script is referred as drive by download. When user pays visit to such a website, the embedded script downloads the malware onto the victim's machine and then further performs exploits. One such example of malware for Android platform is Android/NotCompatible [37].

## 3.3 Malware Detection Techniques

Several malware detection techniques exists for detecting Android malwares. We can sort them into two basic kinds i.e. static and dynamic, however, one more techniques adds to this list which is machine learning [38]. In certain cases, a hybrid analysis comprising of both static and dynamic detection techniques is employed which often yields better results. We discuss some of these in this section.

### 3.3.1 Static Analysis

Static analysis relies on the source code and signatures of the malware under detection without actually executing the malware application. Static analysis is more scalable and has better code coverage than dynamic analysis. Techniques such as obfuscation and dynamic code loading can easily beat static analysis. Some static analysis techniques employed in static analysis include signature-based, permission-based, API-based, Interaction-based and Dataflow-based detection which are discussed below:

#### 3.3.1.1 Signature-based Detection

Signatures of an Android application are extracted and then compared with the signatures of known malware. Usually a hash/checksum is computed of the malware under analysis and compared with the hashes of known malware. These hashes are stored in a signature repository. This signature repository needs to be constantly updated to include the signatures of new malware on a day-to-day basis. Otherwise, the database will become obsolete and new malware variants can easily bypass this detection technique. Obfuscated malwares and dynamic code loading can evade this method of detection.

#### 3.3.1.2 Permission-based Detection

AndroidManifest.xml file contains all the permissions required by an Android application. In permission-based static analysis, an application is categorized as benign or malicious based on the set of permissions it defines in its AndroidManifest.xml file [39]. The type and number of permissions an application requests gives an insight into the application's functionality and various methods are used to perform this kind of detection. But it has certain limitations such as it overlooks the source code and working of the benign app and only relies on permission. It might be the case that a malware app uses the same permissions

as that of benign app. In such a case, no red flag will be raised. Also, this method might give false positive about a benign app classifying it as malicious just based on the permissions.

#### 3.3.1.3 API-based Detection

In this technique, analysis is based on APIs being used in the Android application. APIs are Application Programming Interface available in Android SDK. Android provides these APIs to allow developers to interact with the underlying hardware and use them in their applications in a variety of different ways. AMTs scan the code for any malicious APIs and trigger an alarm based on these APIs. These API calls give a good basic insight about the intent of an Android application. However, in case of polymorphic code, this detection method fails to give any useful information about the Android application.

#### 3.3.1.4 Interaction-based Detection

In interaction-based static analysis, AMTs make a decision about an Android application depending upon the type of interaction between API calls. If certain suspicious interaction is observed between different components of the application under analysis, AMTs would mark the application as suspicious. If for example, an application first intercepts SMS and then sends it to a network, then by simply by looking at this interaction, AMTs can make a guess that the app has a malicious intent.

#### 3.3.1.5 Dataflow-based Detection

Dataflow-based detection technique looks at the sources and sinks of dataflow within an application. If, within an Android application, dataflow occurs between suspicious sources and sinks, AMTs will get triggered. For example, if an Android application has a source defined for getting the device ID and a sink defined that sends this device ID to some remote network, then AMT would assume a suspicious dataflow here.

### 3.3.2 Dynamic analysis

This technique requires the execution of Android application either in real or emulated environment. Tracking the flow of sensitive information or collecting the execution traces and based on this information, the app is marked malicious or benign. Dynamic analysis compensates the static analysis failure when faced with obfuscated, encrypted and dynamically loaded code. However, dynamic analysis has less code coverage and is less

scalable. Dynamic analysis can be classified as anomaly-based and emulation-based techniques.

#### 3.3.2.1 Anomaly-based Detection

Upon the execution of application under analysis in a sandboxed environment, logs of the generated system calls are sent to a remote examination server. Here application's behavior is inferred based on the logged system calls. Presence of anomalous behavior marks the application as malicious. This technique along with other detection techniques is used in dynamic analysis to classify the file as benign or malignant.

#### 3.3.2.2 Emulation-based Detection

Emulation-based detection systems are designed in such a way that the antimalware program examining the file is not on the same system used for the execution of the malware. An agentless system is designed so that the malware may not detect the presence of the antimalware tool. In conventional systems, both the malware and antimalware run in the same virtual machine which may inhibit the malware from depicting its true nature after detecting the presence of the detection tool. Yan et al. [40] presented an agentless emulated detection system where malware runs on the virtual machine and antimalware tool runs analysis from outside of the virtual machine.

### 3.3.3 Machine Learning

Using features extracted from known malwares, similar Android malwares are identified. It has two phases: the training phase and testing phase. In training phase, specific features from known malwares are extracted. Based on these extracted features, new similar Android malwares are classified into benign or malicious ones.

# Chapter 4

# Malware Evasion Techniques

In order to avoid detection by AMTs, next generation malwares tend to be evasive. Malware analysis and detection is a cat and mouse game where if malware analysts are always faced with new breed of existing malwares. These new malware can be termed as evasive malware which are more intelligent, environment aware and adaptive to execution environment. Evasion techniques can thwart the precision of malware analysis tools. Such evasion techniques include obfuscation using packing, anti-debugging tricks etc., resigning, disassembling and reassembling, data encoding, call indirections, code reordering, junk code insertion, string encryption, API reflection, resource modification, NOP insertion, code reuse, steganography and concatenation.

## 4.1 Common Evasion Techniques

Some of the common evasion techniques used by Android malware authors are listed below.

### 4.1.1 Obfuscation [12]

It deceives simple methods of string-matching used in signature-based detection by concealing the attack payload of malware.

### 4.1.2 Code Reuse [12]

This exploit legitimate system requests being used by local running legitimate, benign processes as well.

### 4.1.3 Steganography [13]

It refers to hiding the data in another medium like image, without incurring noticeable changes. Steganography involves converting the image into RGB mode, converting data to be hidden into binary format and then replacing the RGB data with the payload data in any one plain. LSB [41] is one widely used technique of steganography.

### 4.1.4 Cryptography [13]

It makes the code unreadable by applying encryption algorithms such as polymorphic XOR etc. The encrypted piece of code is decrypted at runtime. The only resource available to

AMTs for analysis is the decryption routine which we can further obfuscate to achieve better results.

### 4.1.5 Resigned [42]

This technique involves decompiling an apk file and recompiling it using apktool[43], jarsigner[44] and zipalign[45]. Once recompiled, android application is signed with a custom key since developer keys are not available. This technique does not alter the apk file itself but only its hash by resigning the apk with new certificate and hence altering its signature.

### 4.1.6 String Encryption [46]

It refers to encrypting all the strings using different encryption keys. Encrypt string using xor-string encryptor different for each string. In each Android application, a string.xml file exist which contains list of the strings used in the application. This method encrypts those string names to random/dummy values, hence rendering AMTs unable to detect the malicious application based on string names.

### 4.1.7 API Reflection [42], [46]

API reflection refers to analysis and modification of Java APIs at runtime. Using Java reflection API, static method calls are transformed into reflection calls hence hiding the API calls. Every method call is transformed into a call to that method via reflection. Hence static analysis becomes useless on such method.

### 4.1.8 Resource Modification [46]

This technique involves modifying resource related files. Modifying images in the resources section of the apk file and resource related xml files' data modification hence transforming the identification markers for AMTs rendering them useless against such transformations.

### 4.1.9 NOP Insertion [42]

A no-operation instruction (NOP) is inserted at random into the source code to change both the hash/signatures and delay the execution time.

### 4.1.10 Packing [11]

It encrypts malicious DEX file using an Executable and Linkable Format (ELF) [47] binary that only gets decrypted in the memory at runtime and executed using DexClassLoader[48]. This changes the structure and flow of the APK file.

### 4.1.11 Disassembling and Reassembling [49]

We can disassemble and reassemble the compiled Dalvik bytecode found in the classes.dex file. Components like classes, method and strings etc in a dex file can be arranged in a number of different ways. As a result, each such combination yields a different compiled version of one application. This thwarts the analysis based on signatures of whole classes.dex and also the signatures that look upon the arrangement of components in the classes.dex file.

### 4.1.12 Changing Package Name [46]

The package name which acts as identification mark for a given Android application is defined in the AndroidManifest.xml file. In this technique, we simply change this identification marker to some other name.

## 4.2 Literature Review

This section reviews the evasion techniques with respect to their (i) pros and cons, (ii) evasion tools employed and (iii) detection mechanisms to thwart these evasive techniques.

Mystique [50] is a malware generation framework that uses gene crossover and mutation techniques to generate evasive malwares. Mystique-S, a variant of Mystique, is focused on malware specific to financial charge, phishing and extortion cases [51]. It gathers client's data, delivers the malware at run time and can be evaluated on real devices rather than virtual emulators.

Using genetic operators on existing malware, Sen, Aydogan and Aysan. [52] developed an effective attack with evasion capability that challenges effectiveness of most successful security solutions. Sen et al. also provides a Genetic Programming (GP) based malware detection system incorporating static features of Android applications, which proves very effective against known attacks. However, this technique can only run the malware for limited time period and if run for a long time can trigger analysis of the malicious code.

Rastogi, Chen and Jiang [53] developed DroidChameleon[54] that applies various transformation techniques on malware samples and audits ten popular mobile AMTs being vulnerable to these transformations. However, such evasion is not very effective owing to signature-based detection paradigm.

Zheng, Lee and Lui [55] developed ADAM that employs obfuscation and repackaging techniques like repacking, assembling/disassembling, string encoding, code reordering, junk code insertion, and renaming identifiers, but ignores sophisticated ones such as payload and native code encryption, array data encoding, reflection and bytecode encryption.

Preda and Maggi [56] proposed an Automatic Android Malware Obfuscator (AAMO) to obfuscate exhaustive datasets of Android malware using both existing and new obfuscation techniques. It uses 1,260 malware applications from Genome repository, 6 state-of-the-art AMTs and 17 obfuscation techniques simple and advanced control-flow based modifications, resource renaming and encryption.

Badhani and Muttoo [57] developed eight different evasion techniques to hide malware inside an image of a wrapper Android application using obfuscation, concatenation, steganography, cryptography and their combinations. 402 malware samples developed as a result of the above-mentioned techniques and installed on the real Android devices were then tested against 10 AMTs from Google Play Store.

Chua and Balachandran [58] presented a detailed framework with obfuscation techniques like switch function, method overloading, try-catch function and opaque predicate. The new malware variants retained their malicious operation, thereby indicating that AMTs listed on VirusTotal[59] do not build resilience against obfuscation techniques but only update their signature database to counter malware variants.

RealDroid [60] highlighted a broad range of techniques to evade dynamic analysis in virtualized environments. A set of repackaged malwares with developed heuristics incorporation almost evaded all malware analysis services deceiving numerous analysis tools such as DroidBox[61], DroidScope[62], TaintDroid[63] and online services namely Andrubis[64], SandDroid[65] and TraceDroid[66].

A comprehensive analysis of top 30 AVD (Android Virus Detectors) is conducted in [67]. Vulnerabilities related to AVD malware scan (malScan) are exploited by proposed evasion techniques based on Fast Fourier Transform (FFT) [68] and signal steganography. It works by identifying the scanning period followed by subsequent malicious actions.

A mechanism to evade Android automated runtime analysis is proposed by Diao, W. et. al. [69] using close monitoring of the interaction patterns and events triggered on target device as it differentiates between a human user and an analysis environment. It gives an insight on the efficacy of current dynamic analysis platforms, and could be used in integration with Android malware to monitor the system events before the execution of actual malware.

Albertini and Aprville in [70] demonstrated how one can hide malicious apps inside images using a combination of steganography and cryptography. Using Angecryption, it possible to embed imperceptible, valid and executable bytecode in a benign looking app, and successfully evades static analysis such as disassembly. The wrapping app is installed on the target device, malicious app encrypted into a valid PNG image placed in the assets section of the wrapping app, is decrypted into the payload app and installed at runtime. However, it works only on Android 4.4.2 and not on any later versions.

AVPass[46], another tool developed to automatically bypass Android malware detection systems, offers several obfuscation techniques. It also infers detection features of AV engines and using imitation mode, prevents the code leakage. Imitation mode refers to where query to AV engines is performed in such a way that the actual application sample under analysis is never sent to the AV engine, rather a similar code with selected features is uploaded. AVPass provides an insight into the detection architecture of Android AMTs. Its limitation is that it only bypasses static analysis.

Existing research work has been summarized in Table 3.

**Table 3: Summary of Malware Evasion Techniques on Android**

| Malware Type | Evasion Technique(s) | Pros | Cons |
|---|---|---|---|
| Privacy leaker | Obfuscation (control-based, data-based, both) | Maximizes no. of attack behaviours, minimizes detection | Only audits dynamic analysis based AMTs |

| | | | |
|---|---|---|---|
| Dynamically assembled and loaded malware | Mystique-S- a service-oriented tool | Mystique-S developed malware that are undetectable in case of offline detection | Dynamic Analysis Tools (DATs) can detect dynamically loaded malicious code |
| Malicious Android application (apks) | Genetic Programming (GP) | Most successful AMTs, can be evaded via GP's attack patterns. | Application limited to few malwares, ignores dynamically loaded code |
| Root exploit, information exfiltration, SMS Trojan, dynamic code loading | Repacking, disassembling reassembling, renaming identifier, package name, call indirections, data encoding/ reordering, junk code insertion, payload/ byte code encryption, and composite transformations. | Can evade almost all anti-malware tools | Only thwart static analysis, and not dynamic analysis, Ignores code-level transformations. |
| Credentials stealer, adware, spyware | Repackaging, obfuscation | Can evade Anti-malware tools with little effort | Less comprehensive transformations, lacks composite obfuscation. |
| Genome Malware dataset (AAMO) | Obfuscation (Android specific, simple/ advanced control-flow, resource renaming/ encryption) | Uses sophisticated/ automated obfuscation techniques to evade top AMTs (Avast, Norton), is open source and reproducible. | Only evades scan-time static analysis |
| Spyware, ransomware, banking Trojan | Code reordering based obfuscation techniques- Method overloading, switch or try-catch functions, opaque predicate | Decreased detection rate by 50%, employs updated malware samples that retains its malicious operation | Evades signature- based detection only and uses code reordering obfuscation technique only. |
| Data extortion, root exploits, bot activity and SMS Trojan | RealDroid - Static, dynamic and hypervisor level heuristics disguise | AMTs failed to infer malicious behavior of new malwares. Also, no tool detected VM evasion | Analysis services lacking support for native execution couldn't be evaded. |
| Genome malware dataset | Fast Fourier Transform, signal steganography-based evasion | Exploits malware scan and engine update's null-protection window and is effective | Lack of new malware dataset used for evasion |
| Malicious Android Applications | AngeCryption- encrypting apk to valid PNG and embedding into a benign looking wrapping apk | Makes it possible to embed undetectable, valid and runnable bytecode in a benign looking apk, successfully | Works only on Android 4.4.2 and not on latest Android versions |

| | | | |
|---|---|---|---|
| | | evades static analysis such as disassembly | |
| Malicious Android Applications | AVPass- automatically bypasses AMTs using both obfuscation and inferring detection rules for AMTs. Obfuscation modules include string and variable encryption, API reflection, Resource modification etc. | Bypasses AMTs and gives a good insight about the detection rules of AMTs using inferring and imitation mode. | Bypasses only static analysis and certain features such as inferring AV features doesn't work. |
| | | | |
| | | | |

The aforementioned evasion techniques were successful in bypassing most reputed security solutions, and deceived dynamic runtime detection by analyzing sandbox environment. Mystique and Mystique-S provided reasonable evasion in case of offline detection and addressed privacy leakage and dynamically assembled and loaded malware. However, Mystique audited only one Dynamic Analysis Tool (DAT) and is less effective. Mystique-S too failed to evade when dynamically loaded malware were subjected to DATs. GP based evasion tool claimed to evade most successful AMTs, however, it lacks dynamically loaded code features.

Among the evasion approaches discussed, DroidChameleon, ADAM, AAMO and the system proposed by Badhani and Muttoo are used to test the efficacy of the current AMTs being used for detection of Android malware. Trivial obfuscation techniques developed by DroidChameleon successfully thwart static analysis but fail when DATs are employed for detection. ADAM and RealDroid proposed both evasion and detection frameworks. RealDroid fails to detect VM evasion. ADAM provides reasonable evasion by repackaging and obfuscation but lacks composite obfuscation techniques. AAMO provides exhaustive obfuscation techniques and is flexible in terms of its application but fails to evade DATs.

# Chapter 5

# Proposed Framework for Auditing Android AMTs Using Malware Evasion Techniques

To audit the detection efficacy of known antimalware tools against simple yet sophisticated evasion techniques, a simple, resilient and light weight methodology has been proposed in this chapter and illustrated in Fig. 1. It is based on application obfuscation and dynamic code loading.

The proposed methodology will bypass static analysis in a series of steps owing to the fact that layers of obfuscation will change the application's signatures/hash to a level where it won't be detected by most AMTs. The proposed method consists of three basic evasion modules followed by an auditing module. The three evasion modules are (i). Repacking Module, (ii). Obfuscation Module, and (iii). AngeCryption Module. These modules will be further described in the next section. This technique is simple and lightweight, yet resilient in achieving good evasion results and shedding light on the detection capability of well-known AMTs. The evasion module when implemented alone do not yield better results. However, when taken together, the evasion modules decrease detection efficacy iteratively at each step and the final outcome has an evasion capability to an extent that is incredible. The auditing module simply uploads the resultant application to VirusTotal, an online repository of numerous AMTs. The Fig. shows the framework for evading AMTs.

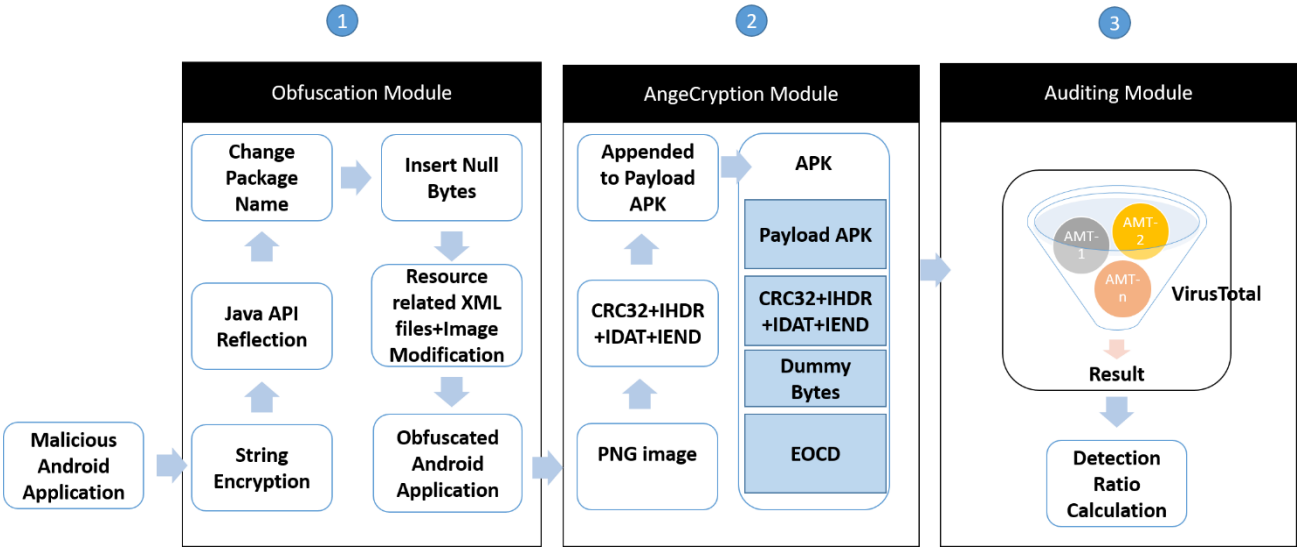



**Figure 5: Proposed Framework for Auditing Android AMTs**

Before delving deeper into the working of the proposed framework, we first describe its individual components in order to provide a comprehensive and thorough understanding of the framework.

## 5.1 Components of the Auditing Framework

Described below are the basic components the proposed framework.

### 5.1.1 Evasion Model

Two evasion modules are used in this framework. These evasion modules are selected on the basis of the simplicity of their implementation, degree of evasion achieved and their interoperability. 'Interoperability', here, refers to the fact that the evasion modules when implemented together in a certain sequence operate successfully and the resultant application does not lose its malicious intent and is working properly on the real Android device. Moreover, these evasion modules are developed by using some existing projects on GitHub and extracting components that function properly on latest Android versions. For example, the first module, AVPass, consists of three components. Only its first component, the obfuscation module works fine. The other components such as inference module, doesn't operate properly at all. Hence, we have taken only the first component for this project. In case of the second module, AngeCryption, its complete project works only on older Android version 4.4.2 and doesn't operate on later Android versions, since the bug it exploits was fixed in later Android versions. However, if we do not implement the entire project as it is,

rather restrict its implementation before the final step, we can make it work on later Android versions. These modules are described in detail in the next subsections. These evasion modules achieve almost 60% evasion.

The proposed framework is implemented as two phase evasion model. In the first, phase, each evasion module is implemented individually on each Android malware sample resulting into a new malware sample. In second phase, two modules are implemented in a specific manner on the same malware applications yielding a new malware sample. This second phase yields the best results.

#### 5.1.1.1 Obfuscation Module

The obfuscation module used in this framework transforms any Android malware into a form that bypasses AMTs. The module's name is AVPass[46]. The module performs apk obfuscation with more than 10 modules. It applies a layer of obfuscation onto the malware so that it evades maximum AVs. According to [46], most of the AVs were bypassed with 3.42/58 (5.8%). 5 strong, 3 normal and 2 weak impact features of AVs were discovered. Also, about 30% bypassing rule combinations are discovered. AVPass has three phases in which it claims to achieve 100% evasion. Under these three phases, AVPass aims to avoid API-based, dataflow-based, interaction-based and signature-based detection. The three phases are as follows:

i. In first phase, individual features of Android binary are obfuscated employing techniques such as string encryption, API reflection, resource modification etc. We can apply these obfuscations in a number of different ways suiting our needs. We can apply them as individual obfuscations or can apply some or all in a specific sequence.

ii. In this phase, features and detection rules of AVs are inferred based on the results of phase 1. The AMTs which detected the malware application in the first phase, its detection features and rules are then inferred and stored.

iii. In the last phase, malicious Android applications are obfuscated in such a way that it evades maximum AMTs. This obfuscation uses the features and rules inferred in the second phase. Based on these inferred rules, obfuscation aiming to remove, hide or

transform the malware apk in such a way that the AMTs is fully bypassed. This feature tends to reduce the number of obfuscation features being applied based on features inferred, applying only those obfuscation necessary to evade analysis.

We limit AVPass implementation only to the first step as the second step doesn't function properly after several trials two steps and since third step is dependent on the second, hence we are forced to use only the first step. Also, we alter the implementation method of AVPass as depicted in [46] in a fashion so as to achieve much sound results even better than with less complexity. We combine the results of this first step with our second module and achieve almost 100% evasion. Moreover, our technique is less complex and is flexible enough to be operable on all Android versions.

We now list the obfuscation components used in AVPass and how we tailored them to our needs to achieve maximum evasion.

#### 5.1.1.1.1 API obfuscation

This obfuscation component provides a list obfuscation utilities for obfuscating APIs and is for evading API-based detection. The utilities provided by this component are injecting random perturbations, injecting API between two existing APIs, listing APIs, injection of API between specific points, modifying package and file name, removing all permissions and inserting benign permissions. In order to break API-based detection, we can either inject dummy APIs or modify all family/package names. The number of APIs to be injected depends upon the size of the malware, the bigger the malware, the more number of APIs need to be injected and vice versa.

#### 5.1.1.1.2 String and Variable Encryption

Variables in an application are encrypted using simple caesar cipher while strings are encrypted using xor-string encryptor which is different for each string. This is done by inserting massive number of getStr() functions. This kind of obfuscation breaks Signature-based detection. This method encrypts those string names to random/dummy values, hence rendering AMTs unable to detect the malicious application based on string names.

#### 5.1.1.1.3 Package, Class and Method Obfuscation (PCM)

This obfuscator obfuscates package and class/methods present in an Android application. It changes package name, class/method name in such a way that it no longer serves as a signature for detection. It also modifies AndroidManifest.xml files modifying the main package name (package=”com.a.b.c”), modifying components name such as activity and services etc. In case of class name, it first checks whether a particular class name exists, modifies line classes (L class) except for real class names. Also it encrypts the references to class names if found in the xml files. Moreover, it scans all xml files found in res section of Android application and change all class references to encrypted form. PCM also has a file/directory name changer which modifies file name which is actually the file name provided internal definitions and its references are modified and changes directory name also.

#### 5.1.1.1.4 Java API Reflection

In this obfuscation component, reflection is performed for each file by generating a set of wrapper functions. The wrapper function contains the actual payload/malicious function. These set of wrapper functions can be either stored in an original smali file, in a separate file in the same directory or in any one specific wrapper file. We can generate these wrappers either for each API call or for a set of same API calls in each file/package/method etc. This techniques uses the Java API for reflection.

#### 5.1.1.1.5 Resource Obfuscation

Resource obfuscation modifies the contents of ‘res’ directory. It modifies images/swf, data in resource related xml files, nullifies payloads (.so, .jar, .zip, &c) and removes “unknown” directory. It performs some of the same functions as don PCM component such as modifying class names, modifying class name references in xml files. It modifies images in the res section by changing images’ hex values. This is done by either modifying pixels or adding one byte. It also alters the string, id and drawable in XML files. The nullify payload function renders the application useless, hence is not recommended.

### 5.1.1.2 Angecryption Module

Angecryption [70] is the second module after AVPass that we use in our proposed solution. It works on the idea of encrypting any given input into any JPG or PNG image. Its details have already been discussed in the section of chapter .However, we give its technical details

here. AngeCryption does not exactly encrypts the input file into the image rather it transforms it into something that looks identical to the image file[Ange].For the sake of understanding, we precisely define the PNG image format here. A PNG file consists of the following parts:

- File Header: An 8-byte fixed PNG signature which reads '0x89 PNG 0x0d 0x1a 0xa'. This signature helps in the identification of PNG file as a valid PNG file.

- Garbage Chunk: A chunk is comprised of chunk length (4 bytes), chunk id (4 bytes), chunk data and CRC32 of chunk data and id. The data residing here is usually ignored by image reading tools.

- Header Chunk: It is mandated by PNG specifications that a header chunk initiate a PNG image.

- Data Chunk: The actual image data blocks reside in this section.

- End Chunk: This is the end of file marker and terminated the PNG.

AngeCryption uses AES as the encryption method where a single AES block equals 16 bytes. To generate the output as desired by AngeCryption, a suitable IV is first selected. The first cipher block C1 needs to be equal to the PNG file header (8 bytes) + chunk length (4 bytes) + chunk id (4 bytes). This, coincidentally, equals 16 bytes fitting perfectly an AES block. Now, the plain text is input from the payload application. IV is selected in such a way that it equals AESk^-1 (Ci) +Pi. IV, here, is selected in such a way that it yields the first cipher block desired whereas in real encryption, IV is random. After the appropriate selection of IV, a modified apk is generated. Modified here refers the payload apk with appended data at the end. The data appended at the end is constituted of decrypted CRC32 checksum + payload image file blocks +end chunk. The reason for appending this data is to generate original data when encrypted data is decrypted i.e. AESk (AESk-1(Pi))=Pi. The important feature of AngeCryption is that it is independent of the encryption method employed and the kind of payload format used i.e. we can use either PNG, JPG image or a PDF or FLV file. Moreover, AngeCryption ensures that selecting an appropriate IV, we can generate desired cipher block,

source format can tolerate some appended data and header+chunk declaration data of the source format fits in the block size.

In order for the AngeCryption tool to convert an input file into a target image, we need to modify the input file in such a way that its content remains intact. The parameters given to this tool include

- input file which needs to be encrypted into a target image,
- target image that is what the input file needs to look like,
- modified input which is a modified version of the input file manipulated to be handled by the angecrypt tool, as the input file in its raw form cannot be encrypted to a target image
- Key which is used for encryption,
- Encryption algorithm. Angecryption supports AES128-CBC and 3DES-EDE2-CBC

The output from the tool is a modified input file and a generated Python script containing the required IV. We halt this process until we get the modified input file and do not encrypt this to the target image for reason that this final image product does not work on latest Android versions. This conversion of payload apk to a modified apk is illustrated in the Fig. 6. The modified apk is our end product out of the evasion model.

The complete angecryption tool then embeds the resultant target image into a wrapping apk in the assets section. One can be flexible in this implementation method. This wrapping apk reads the asset, opens the PNG, decrypt it using the cipher used for encryption, write it to the SD card and installs the apk onto the target device.

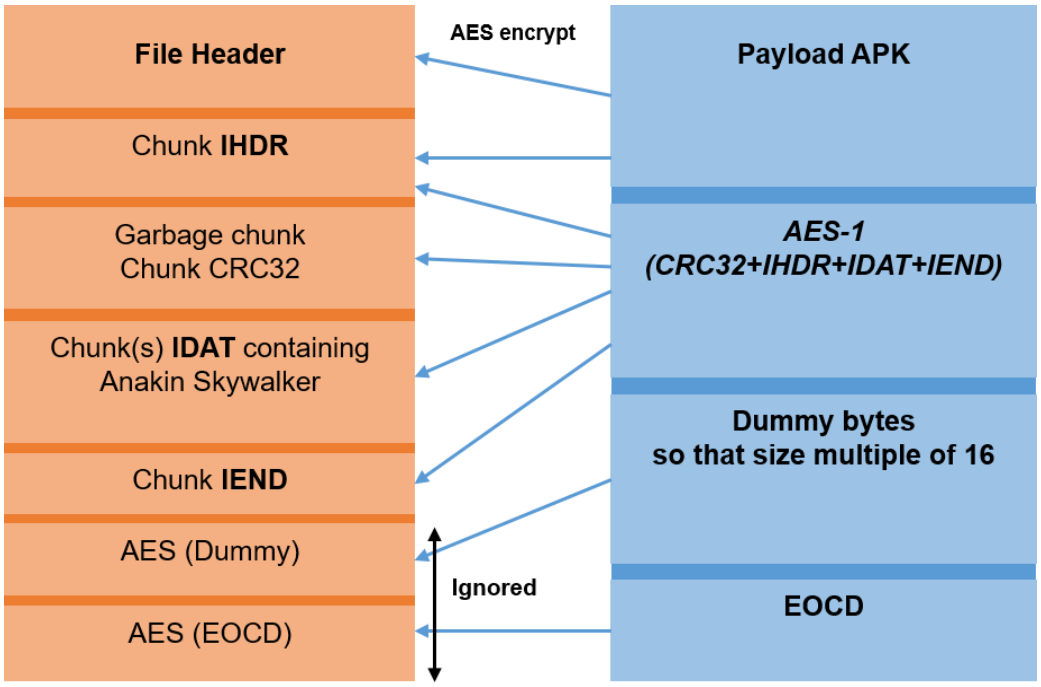



**Figure 6: Layout of PNG and modified APK**

We tailor its implementation to our needs since as mentioned earlier that its complete implementation only works on Android 4.4.2 and fails on later Android versions. We only use the modified apk as illustrated in the figure above. The output of AVPass module is angecrypted and later used for auditing the AMTs. We skip the final encryption to a valid PNG and embedding into a wrapping apk part. Our outcome is an undetectable, valid and runnable apk on the latest Android versions.

### 5.1.1.3 Phases of Evasion Model

We divide our evasion model implementation into two phases. First phase refers to implementation of individual implementation of the evasion module and submodules, the second phase refers to the implementation of all the two modules and submodules in a customized sequence so as to give the best results.

#### 5.1.1.3.1 Individual Evasion Module Implementation

In this phase, each module and submodule is implemented individually and the outcome is used for auditing the AMTs. Obfuscation module AVPass is first implemented as a whole and later its distinct submodules are implemented individually. Also, Angecryption is also implemented individually. The detection results of each are recorded and compared.

#### 5.1.1.3.2 Multiple Evasion Modules Implementation

In this phase, we use the evasion modules and submodules in a customized sequence on the malware sample. The customized sequence developed is based on the best evasion achieved i.e. the sequence or pattern that gives the best results. The resultant sequence yields the final version of the proposed framework as depicted in the Figure.

### 5.1.2 Auditing Model

After applying the proposed evasion model upon the sample Android malwares, we audit the detection efficacy of the some top notch AMTs. For the said purpose, we use a repository of known AMTs. VirusTotal [58] is one said repository where we can upload our malware sample and determine the extent of its malicious intent. VirusTotal uses static analysis approaches as mentioned earlier to classify a given sample. In our approach, we upload the malware sample obtained at each step to the VirusTotal and look for the number of AMTs that detect it as malicious or benign. With each step of our implementation, we observe that the number of AMTs that detect the malware sample decreases.

#### 5.1.2.1 Steps for Auditing AMTs

The Auditing steps are listed below:

i. First of all, we upload the malware sample to VirusTotal in its raw form i.e. without any evasion implementation and in its purest form as it exists. We note down the results of the AV engines that detect the malware. We calculate the detection percentage by dividing the number of engines that detected the malware to the total number of engines used i.e.

Detection Ratio= (no. of AMTs that detected the malware sample/Total number of AMTs used by VirusTotal)*100

ii. We now apply the evasion technique proposed. The output of the evasion model at each interval is uploaded to VirusTotal.

iii. After confirm upload step, VirusTotal shares the sample with various AMTs engines, within seconds we get the number of AV engines that detected the malware sample. We again calculate the detection ratio using the formula mentioned above.

iv. We compare the results of detection ratio obtained for each step of evasion applied and note down the AMTs that detected the malware and those that are bypassed.

v. We also look for signatures used for detection at each step and the type of detection method evaded.

vi. This process is repeated for approximately 1200 malware samples, 200 raw malware samples and around 1000 malware samples obtained as a result of the evasion application.

vii. We determine the best evasion combination and hence the malware samples which bypass most AMTs. Also, we determine the most resilient AMTs against our evasion technique.

viii. At the end, findings regarding AMTs are presented.

These steps are repeated for each evasion module. Using this method helps us analyze the robustness and detection efficacy of AMTs. The AMT should give ideal results irrespective of any kind of evasion applied.

However, as we will notice that application of evasion modules alters the detection efficacy of these AMTs. Malicious files are being classified as benign files after the application of sufficient and appropriate evasion techniques. Certain evasion components and their combination are better detected while others are better evaded.

# Chapter 6

# Experiment

This chapter outlines all the prerequisites including both hardware and software requirements, malware dataset and antimalware engines being audited.

## 6.1 Environmental Setup

We now list the pre-requisites for the implementation of this proposed framework. We use Kali Linux installed as a guest Operating System on VMWare Workstation. VMware Workstation is installed on Windows10 as Host Operating System. Each malicious application undergoes decompilation for the implementation of obfuscation module. Hence, we use apktool. Apktool for its proper functioning needs JAVA Virtual Machine, so JAVA is also a prerequisite for the project. Also, both AVPass and Angecryption are written in Python, hence, Python is also a prerequisite for this project. We use Kali Linux for this project because Java, apktool and Python come preinstalled on it, we only need to take care of the correct version of these software. The Android device used for real time testing is Huawei LUA with Android version 5.0.2. The table below lists down the correct version of these software we used for our experiment.

**Table 4: Software and Hardware Requirements**

| Sr. No. | Software | Version |
|---|---|---|
| 1. | Windows 10 | Windows 10 Pro |
| 2. | VMware Workstation | Pro v12.0.1.3160714 |
| 3. | Kali Linux | kali-linux-2.0-amd64 |
| 4. | Java | 1.8.0_45 |
| 5. | Apktool | 2.3.0 |

| | | |
|---|---|---|
| 6. | Python | 2.7.9 |
| 7. | Android | 5.1 |

## 6.2 Malware Dataset

In order to audit AMTs, we use around 200 Android malware samples collected for GitHub repository [71], a large open collection of Android malware samples collected from various sources and mailing lists. We first applied the evasion model on one Android sample to select the best set of evasion components and later those evasion components are applied on 200 Android malware samples to check the consistency of our results.

We used dendroid malware as our sample file and applied the evasion components. Dendroid is a malware development kit that is used for automating and developing Android malwares [72]. Dendroid is a Remote Access Trojan (RAT) which allows malware authors to develop malwares with features such as intercept SMS message, video recording and audio input, running an application and dialing a phone number. Also, traits such as anti-emulation to help the malware stay hidden from Bouncer, Google Play Store's security system for blacklist malicious apps from being uploaded to the Play Store. We applied the framework in two phases as mentioned earlier. In first phase, we implement individual evasion modules and submodules, check the detection ratio by uploading the malware to the AMTs repository. Also, we select best individual components and use them in the second phase.

**Table 5: List of Individual Evasion Components**

| Sr. No. | Evasion Component |
|---|---|
| 1. | String Encryption (SE) |
| 2. | Variable Encryption (VE) |
| 3. | Random Perturbation (RP) |

| | |
|---|---|
| 4. | One-by-One Perturbation (OOP) |
| 5. | Change Package Name (PN_API) |
| 6. | Change File Name (FN) |
| 7. | Remove All Permissions (RAP) |
| 8. | Insert Benign Permissions (IBP) |
| 9. | Change Package Name (PN_PCM) |
| 10. | Insert Null Bytes (INB) from PCM |
| 11. | Insert Benign Class (IBC) from PCM |
| 12. | Resource Obfuscations (RO) |
| 13. | Angecryption (ANGE) |

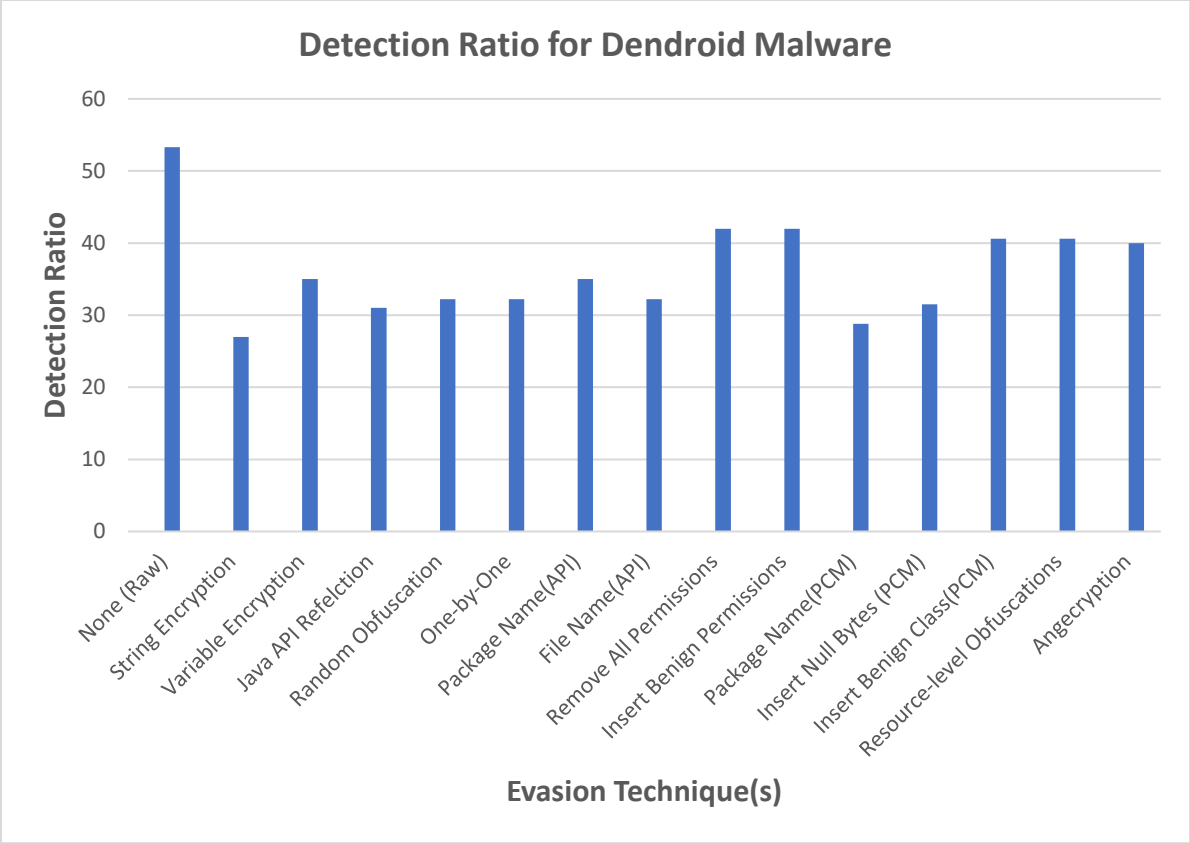



**Figure 7: Detection Ratio for Individual Evasion Module Implementation**

In second phase, using the best evasion components from first phase, we implement multiple evasion components in a customized sequence so as to arrive at a sequence which gives the best evasion results and increases detection complexity for AMTs. We can see that we achieve 0% detection for String Encryption (SE) + Java API Reflection (JAR) + Change Package Name(PN) from PCM module +Insert Null Bytes (INB) from PCM + Resource-Level Obfuscation (RO) + Angecryption (ANGE).

**Table 6: List of Multiple Evasion Components**

| Sr. No. | Evasion Component(s) |
|---|---|
| 1. | String Encryption (SE) +Java API Reflection (JAR) |
| 2. | String Encryption (SE) +Java API Reflection (JAR) + Change Package Name (PN_PCM) |

| | |
|---|---|
| 3. | String Encryption (SE) +Java API Reflection (JAR) + Change Package Name (PN_PCM) from PCM + Insert Null Bytes (INB) |
| 4. | String Encryption (SE) +Java API Reflection (JAR) + Change Package Name (PN_PCM) from PCM + Insert Null Bytes (INB) + Resource Obfuscation (RO) |
| 5. | String Encryption (SE) +Java API Reflection (JAR) + Change Package Name (PN_PCM) from PCM + Insert Null Bytes (INB) + Resource Obfuscation (RO) +Angecryption (ANGE) |
| 6. | AVPass: Java API Reflection (JAR) + String Encryption (SE) + Variable Encryption (VE) + Resource Obfuscation (RO) |
| 7. | Java API Reflection (JAR) + String Encryption (SE) + Variable Encryption (VE) + Resource-level Obfuscation (RO) + Angecryption (ANGE) |

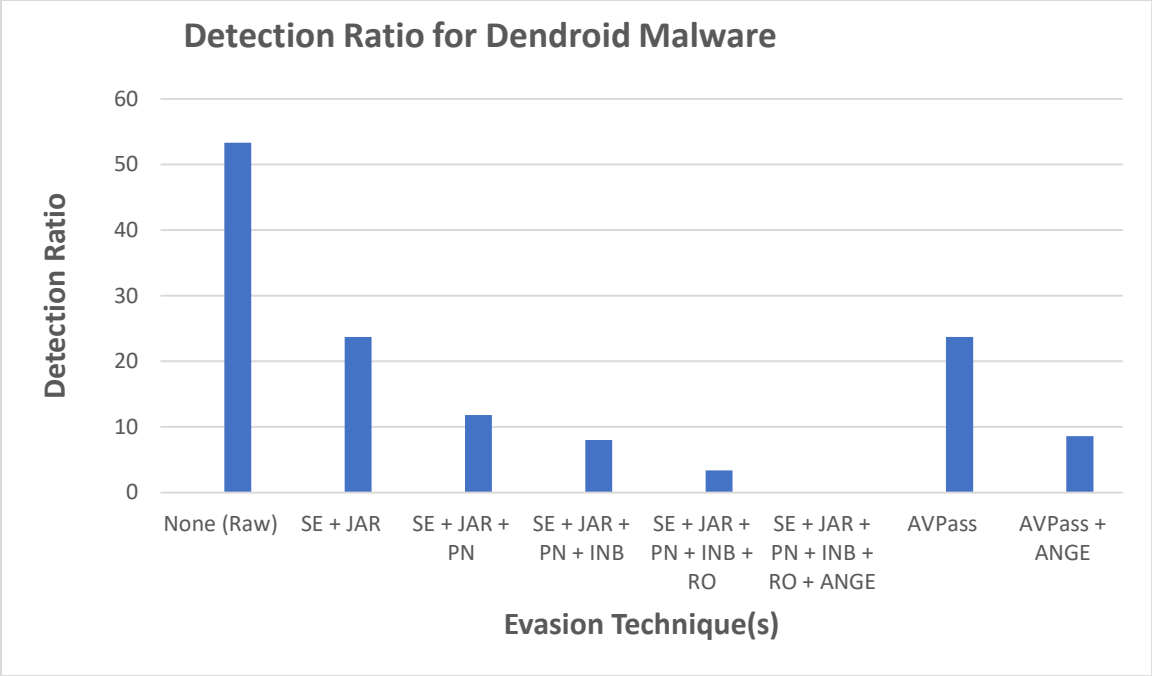



**Figure 8: Detection Ratio for Multiple Evasion Module Implementation**

After selecting best evasion component implementation sequence, we apply it over 200 malware samples. As a result, we obtain more than 1, 000 new malware variants. We see a consistency in our results with that of obtained for dendroid malware.

## 6.3 Malware Detectors

Instead of depending on a single malware detector for drawing results about the detection efficacy of AMTs and the evasion capability of the proposed framework, we use a handful of malware detectors because each malware detector uses different technique for detection of malwares. Relying on a single AMT would either result in excellent detection or very poor detection. Hence, to avoid this shortcoming, we use first a single malware file with the implementation of evasion components to check the detection capability of several AMTs simultaneously. Also, instead of uploading the file manually to different AMTs, we use VirusTotal.

### 6.3.1 VirusTotal

VirusTotal which not only provides more than 60 AMT engines for the analysis of files, URLs, IP addresses, domains or file hashes, but also gives basic details about the file uploaded.

#### 6.3.1.1 VirusTotal Sandbox Integration

Using its integration with three Android sandboxes namely:

##### 6.3.1.1.1 VirusTotal Droidy[73]

It characterizes actions that Android applications perform when installed and opened on Android devices. These sandboxes extract information such as

- SMS related activities
- Network communications including http requests and DNS Resolutions
- File System Actions including files opened, files written and files deleted
- Process and Service Actions including Services started and Processes Tree
- Synchronization Mechanisms and Signals including signals hooked
- Permissions checked, Registered Receivers, Java Reflection calls
- SQLite Database usage
- Crypto-related Activity

##### 6.3.1.1.2 Tencent HABO

Tencent's setup comprises analysis environments not only for Windows, but also for Linux and Android. It can thoroughly analyze malware samples from both static information and dynamic behaviors perspective, trigger and capture behaviors of the samples in the sandbox, and output the results in various formats. It was the first Linux-base ELF files' behavioral characterization engine and among the first sandbox to be integrated with VirusTotal under the multisandbox project [74].

##### 6.3.1.1.3 VirusTotal Androbox

VirusTotal Androbox is one of the sandboxes integrated to VirusTotal under the multisandbox project. It shows some behavioral information regarding the malware apk

under analysis. Androbox shows network communications such as http requests and file system actions such as files opened, deleted etc. performed by the apk.

# Chapter 7

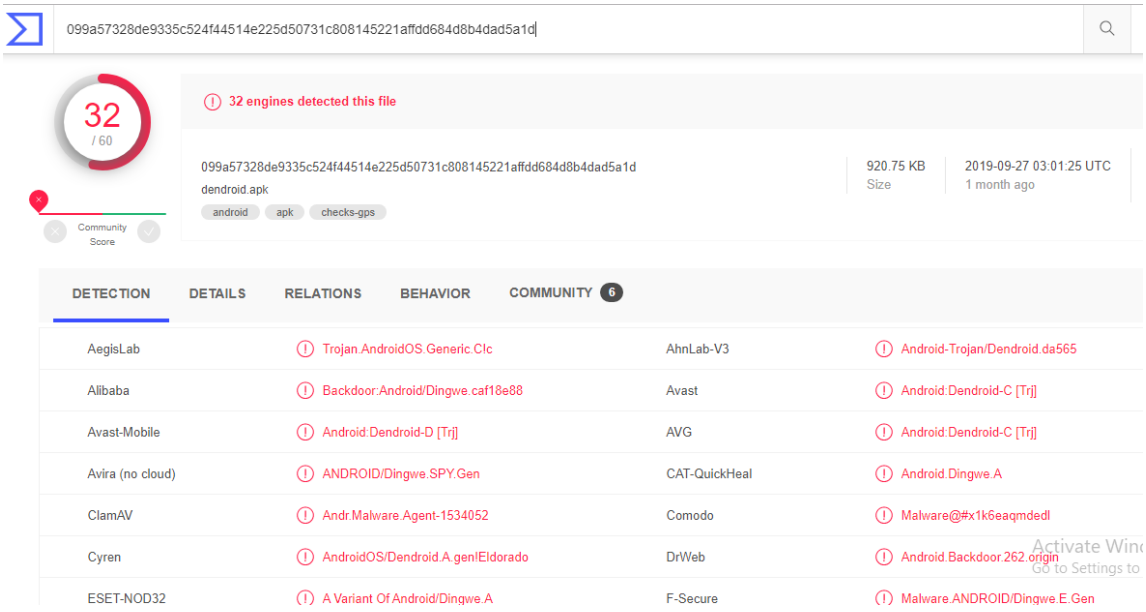
# Implementation Results

The malware app used for testing is Dendroid capable of hiding from the Android emulators, and a sophisticated remote administration tool. At each interval, the application was installed on real Android device to check that it retains its malicious intent and installs successfully. The application did not crash at any moment.

## 7.1 Uploading Malware Variants to VirusTotal

### 7.1.1 Raw Malware

We first upload Dendroid to VirusTotal and check the results. It has a detection rate of 32/60 according to VirusTotal as illustrated in Fig. 9. The detection ratio is ((32/60)*100) = **53.3%**. It must be noted that higher the detection ratio, the more AMTs detect the malware as malicious and lesser the degree of evasion achieved. The aim is to achieve as less the detection ratio possible and based on that determine the flaws present in AMTs. All the known antimalware solutions particularly Avast, AVG, Kaspersky, Symantec and McAfee etc. are able to detect it as malicious file.



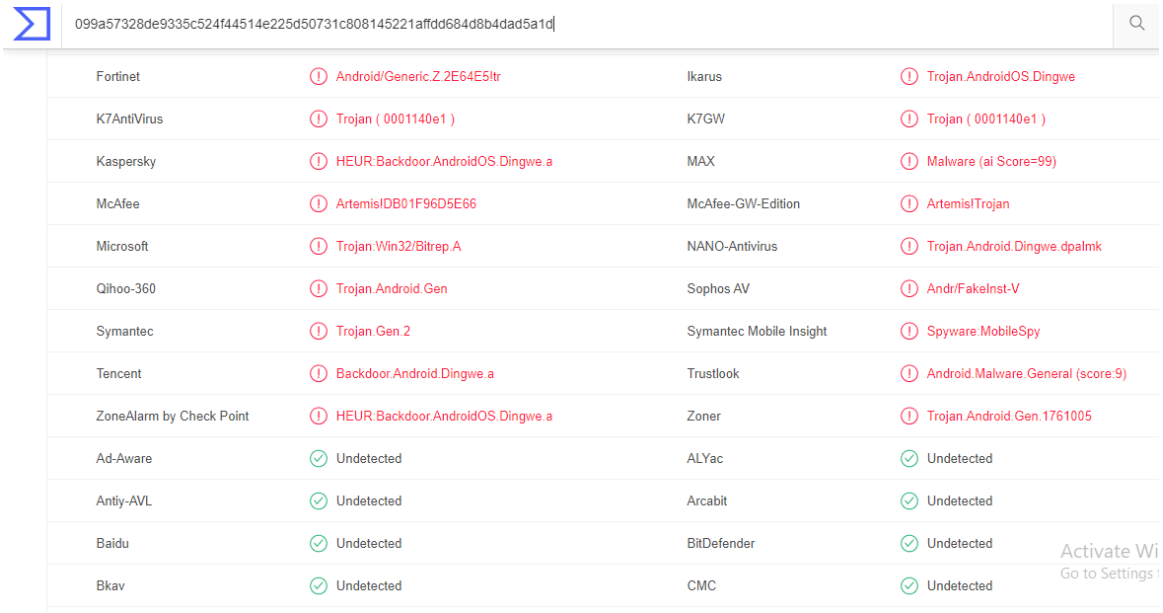



**Figure 9: VirusTotal Result for Raw Dendroid Malware**

### 7.1.2 Individual Evasion Module Implementation

In second step, we start applying the evasion techniques on the malware samples and upload the malware variants to VirusTotal. We first make the malware go through the first phase evasion i.e. application of individual evasion modules and submodules. AVPass and Angecryption are applied individually in this phase. The malware sample goes through the obfuscation of submodules of AVPass and then, Angecryption as mentioned earlier. We apply all the submodules individually.

#### 7.1.2.1 String Encryption (SE)

The Dendroid malware undergoes string encryption. The number of AMTs that detect the new malware variant drops to half. The detection ratio comes out to be 27%. Thus applying only string encryption drops the detection rate to almost half. This obfuscation component evades API-based and signature-based detection.

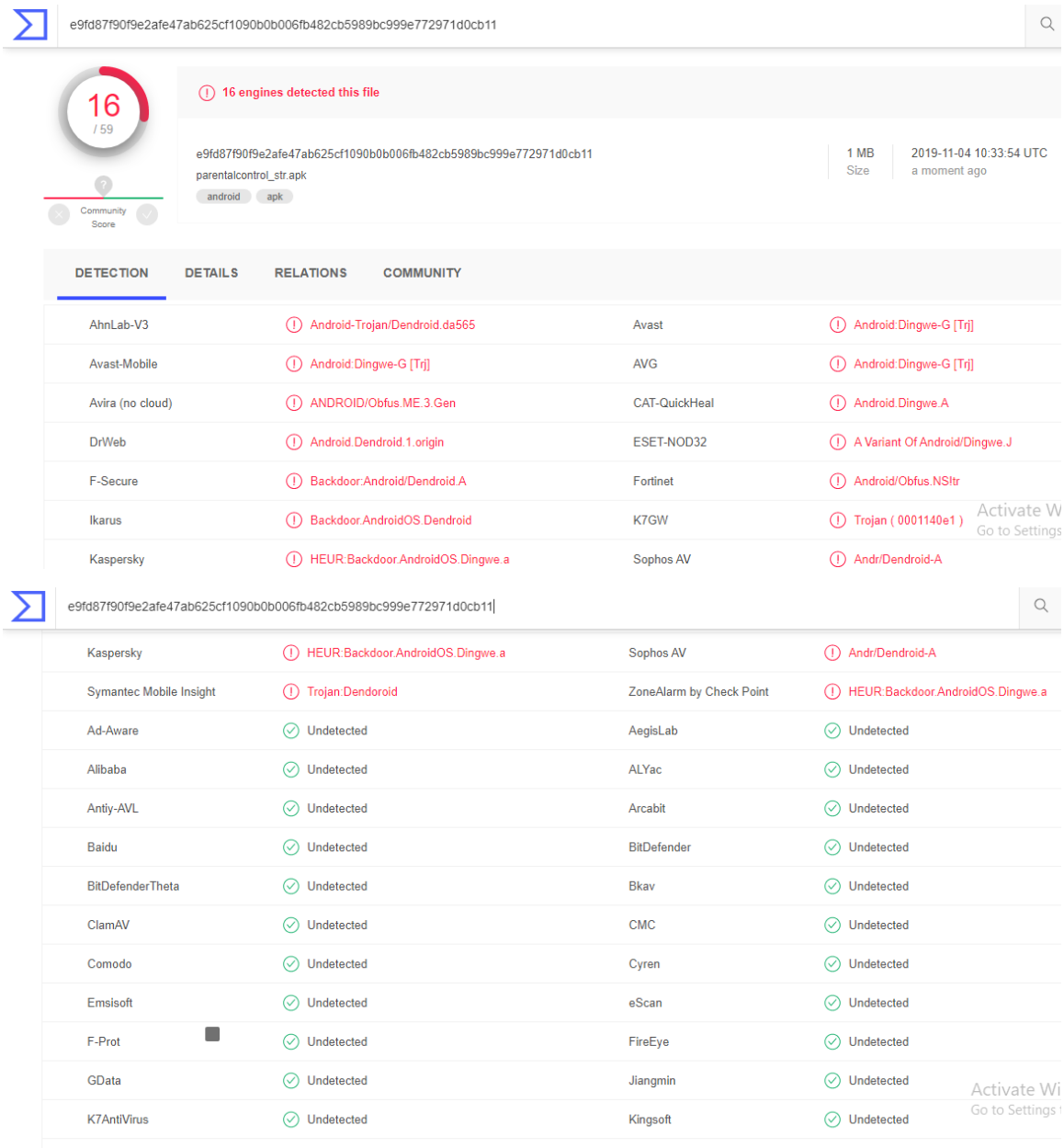



**Figure 10: VirusTotal Result for SE Implemented Dendroid Malware**

### 7.1.2.2 Variable Encryption (VE)

Encrypting variables results into a detection ratio of 35% which is almost 18% decrease in detection rate. Variable encryption also evades signature-based static analysis.

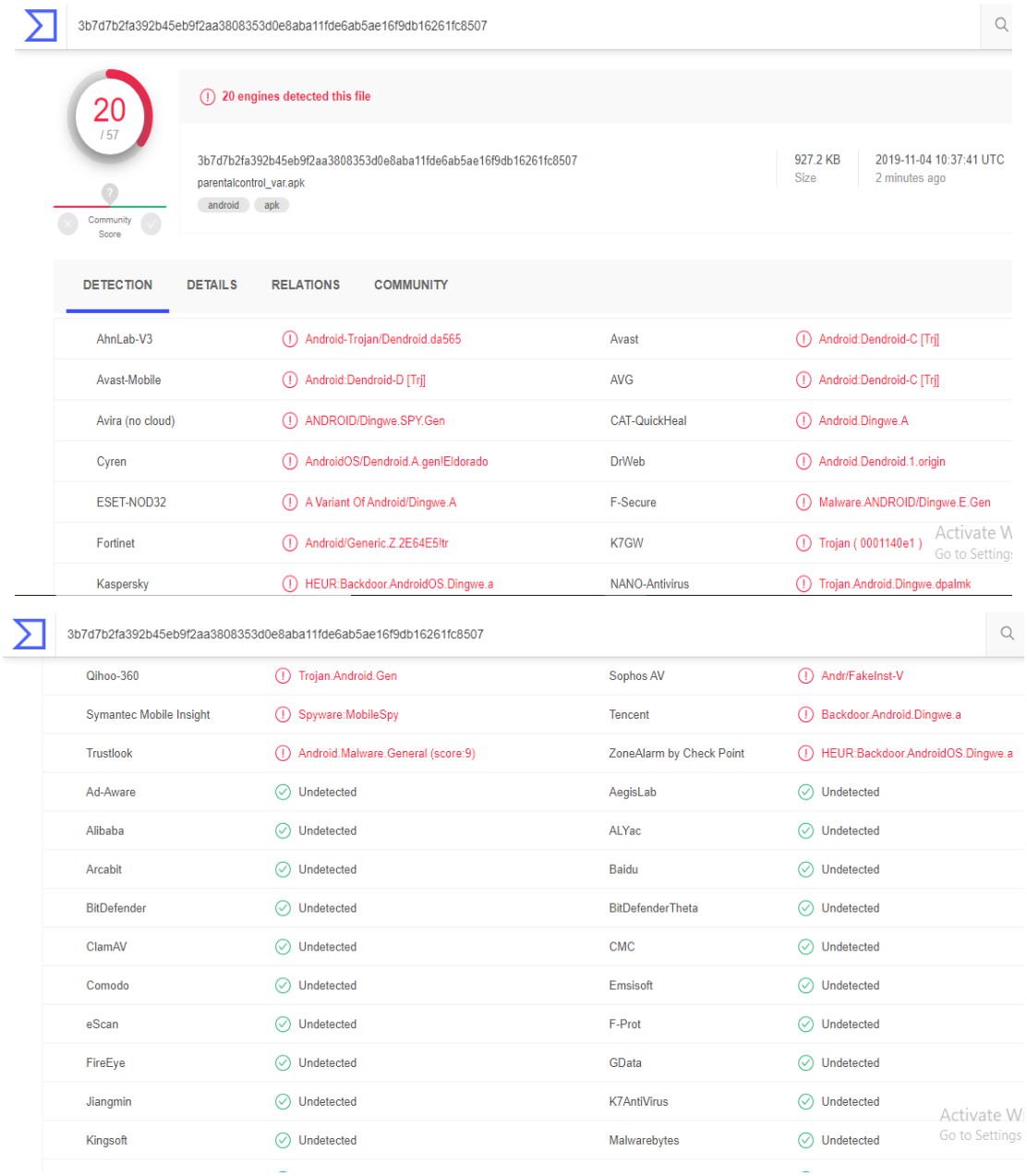



**Figure 11: VirusTotal Result for VE Implemented Dendroid Malware**

### 7.1.2.3 Java API Reflection (JAR)

Applying Java API reflection yields a detection ratio of 31%. 18 AMTs out of 58 are able to detect the malware as malicious. It evades API-based and interaction-based detection.

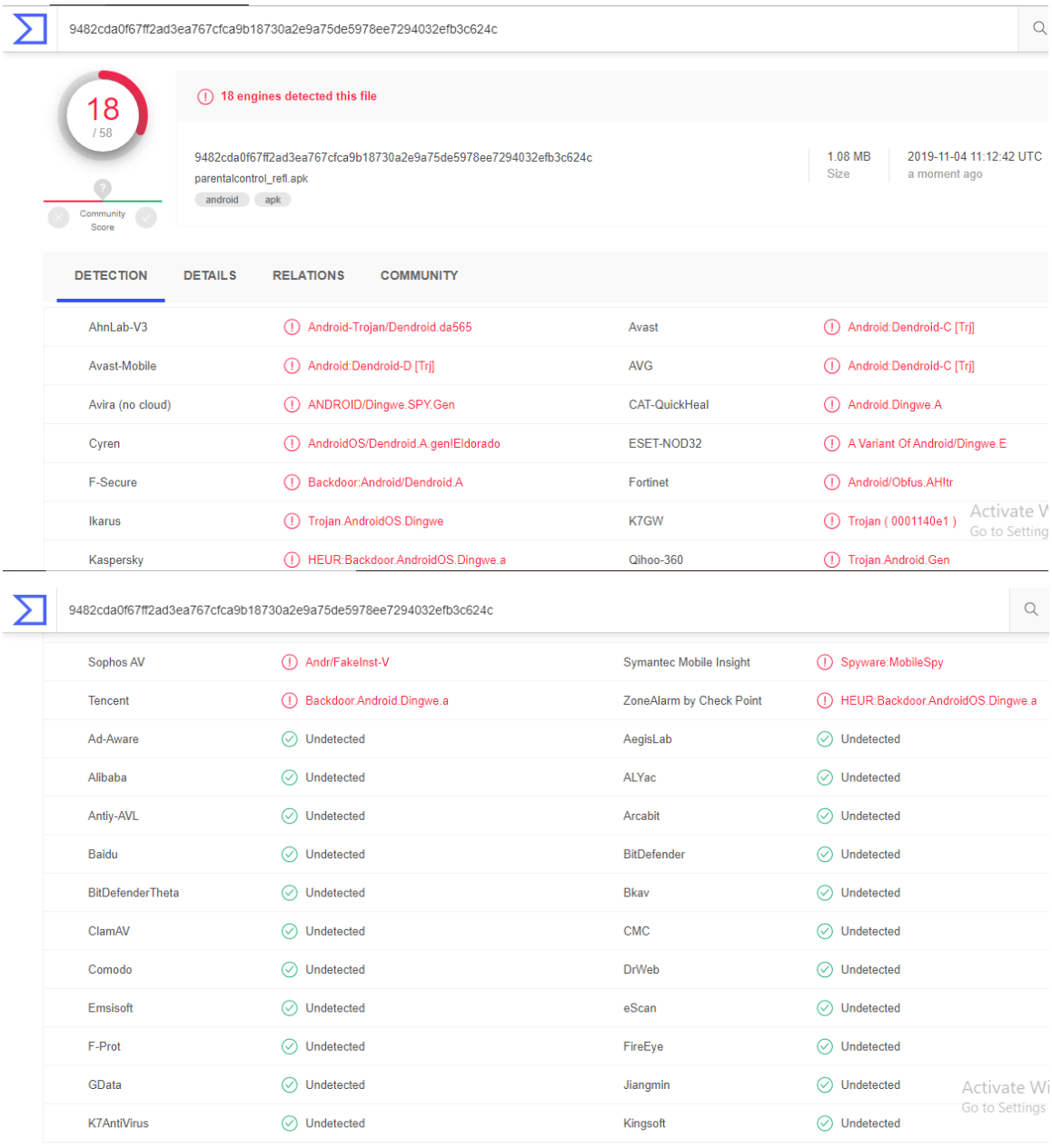



**Figure 12: VirusTotal Result for JAR Implemented Dendroid Malware**

### 7.1.2.4 API Obfuscations

#### 7.1.2.4.1 Random Perturbation (RP)

Random perturbations are performed i.e. APIs are inserted at random. Employing random API obfuscation helps bypassing 13 AMTs out of 32 AMTs that detect raw malware. A detection ratio of 32.2% is achieved. It helps evade API-based static analysis technique.

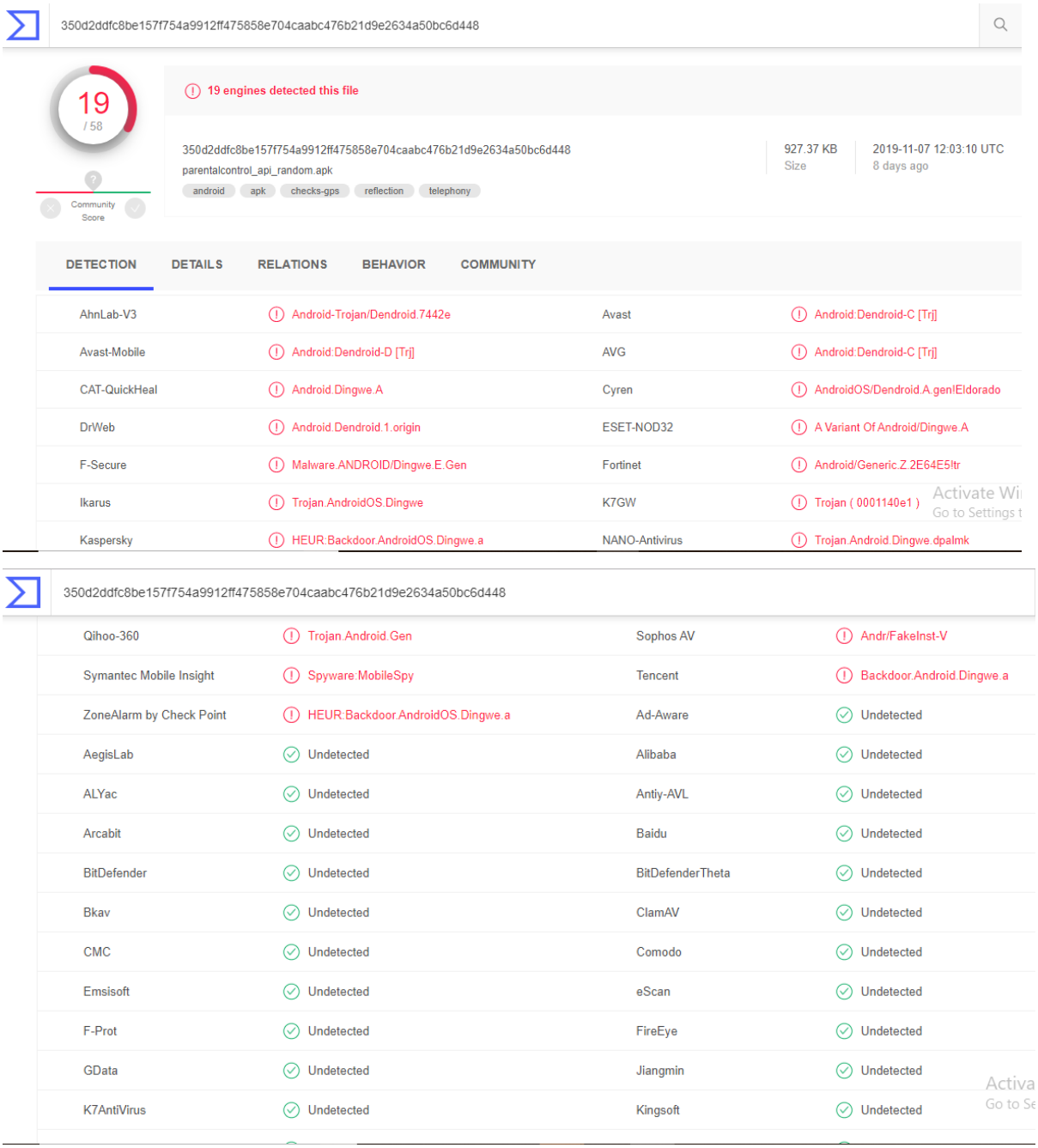



**Figure 13: VirusTotal Result for RP Implemented Dendroid Malware**

#### 7.1.2.4.2 One-by-One Perturbation (OOP)

APIs are inserted between two existing APIs. The detection ratio for this obfuscation component is 32.2%.

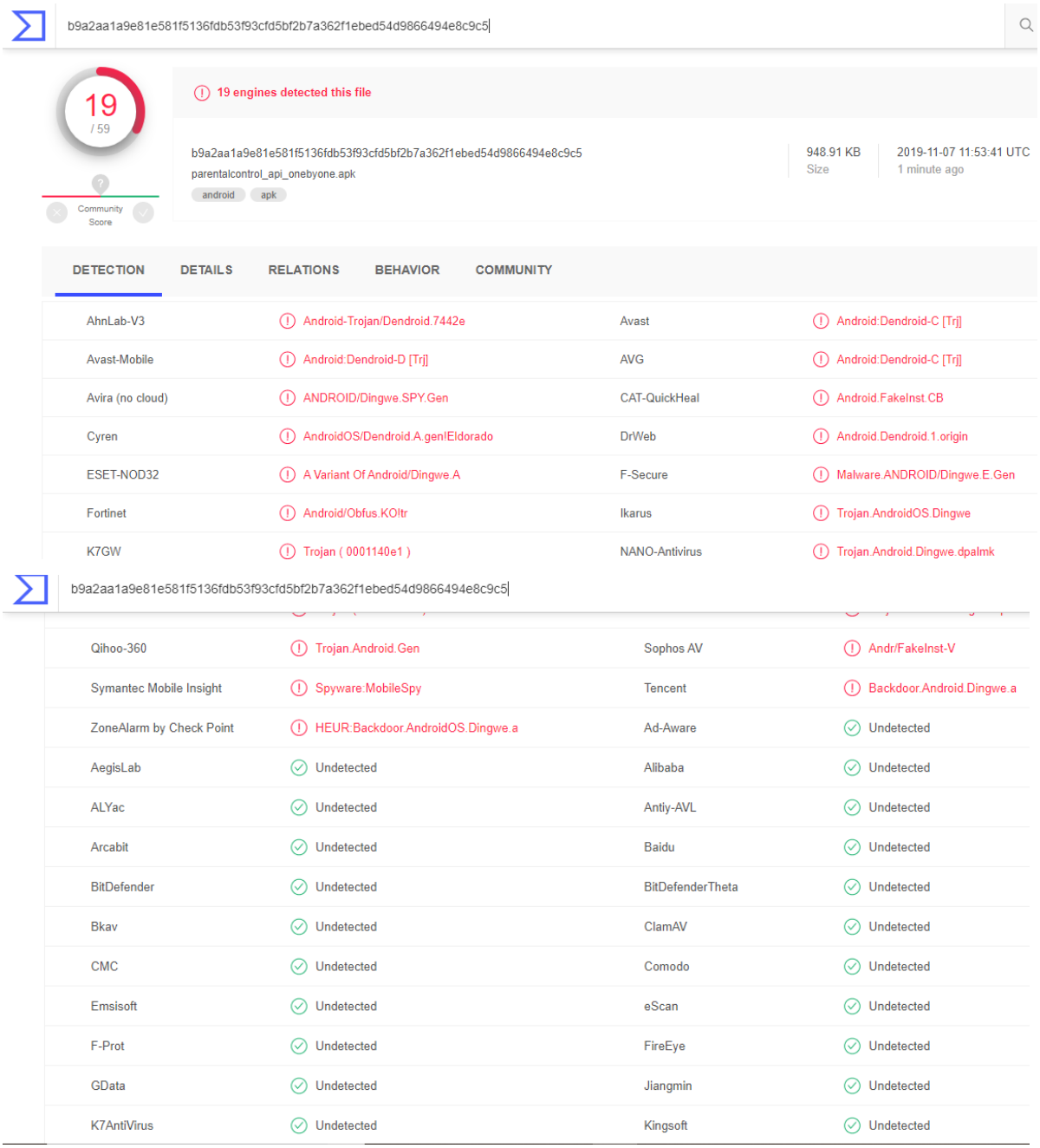



**Figure 14: VirusTotal Result for OOP Implemented Dendroid Malware**

#### 7.1.2.4.3 Change Package Name (PN)

Changing package name brings about evasion of around 11 AMTs. The detection ratio is around 35%. It partially evades signature-based and API-based detection.

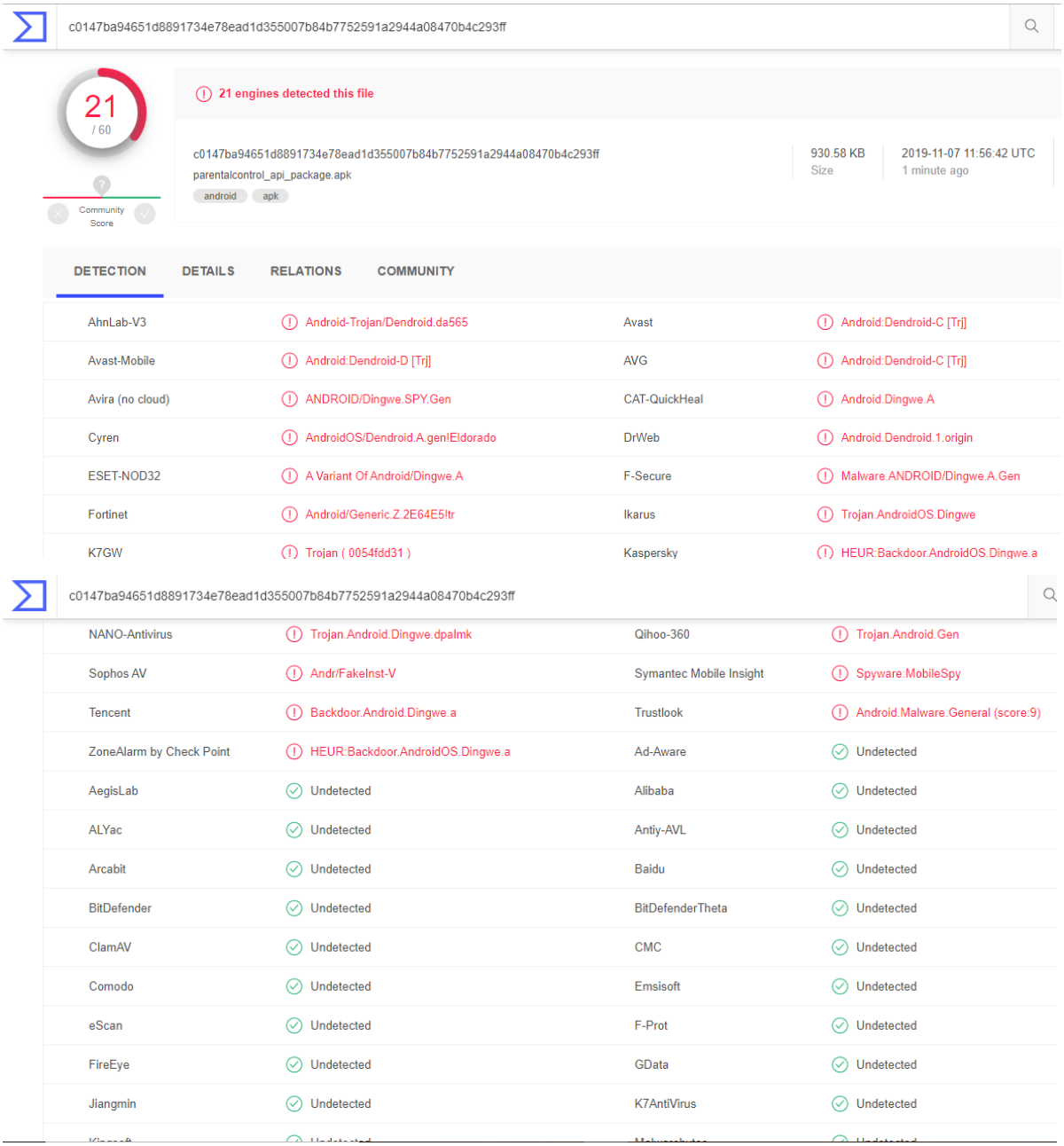



**Figure 15: VirusTotal Result for PN_API Implemented Malware**

#### 7.1.2.4.4 Change File Name (FN)

Changing file name yields 32.2% detection ratio and evades almost 13 AMTs. It also partially evades signature-based static analysis.

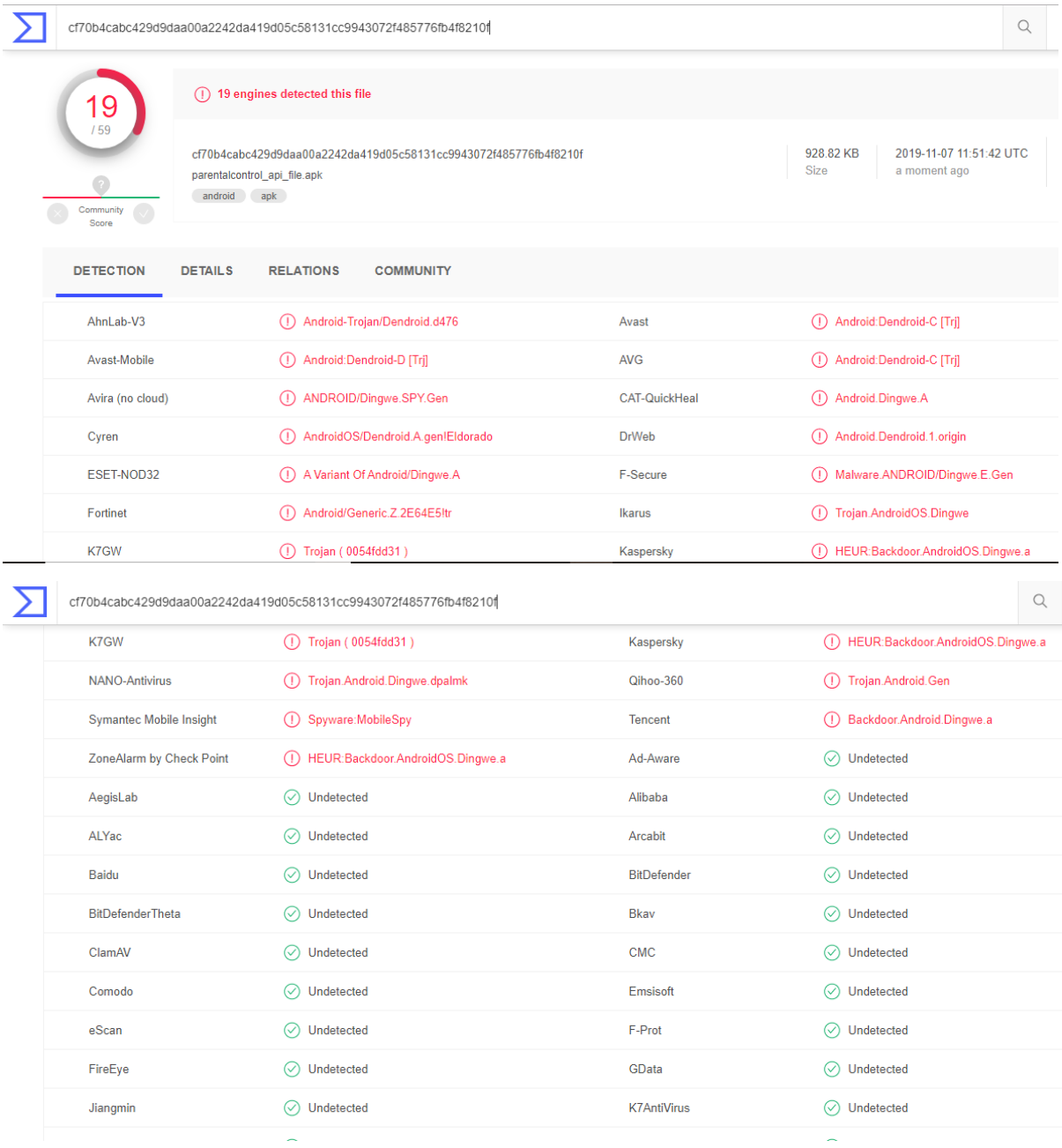



**Figure 16: VirusTotal Result for FN Implemented Dendroid Malware**

#### 7.1.2.4.5 Remove all Permissions

Removing all permissions evades only 7 AMTs and the detection ratio is 42%, much closer to that of raw malware. Though it tends to evade permission-based static analysis, yet as mentioned earlier, this evasion renders the malware useless. Hence, we tend to avoid this obfuscation technique.

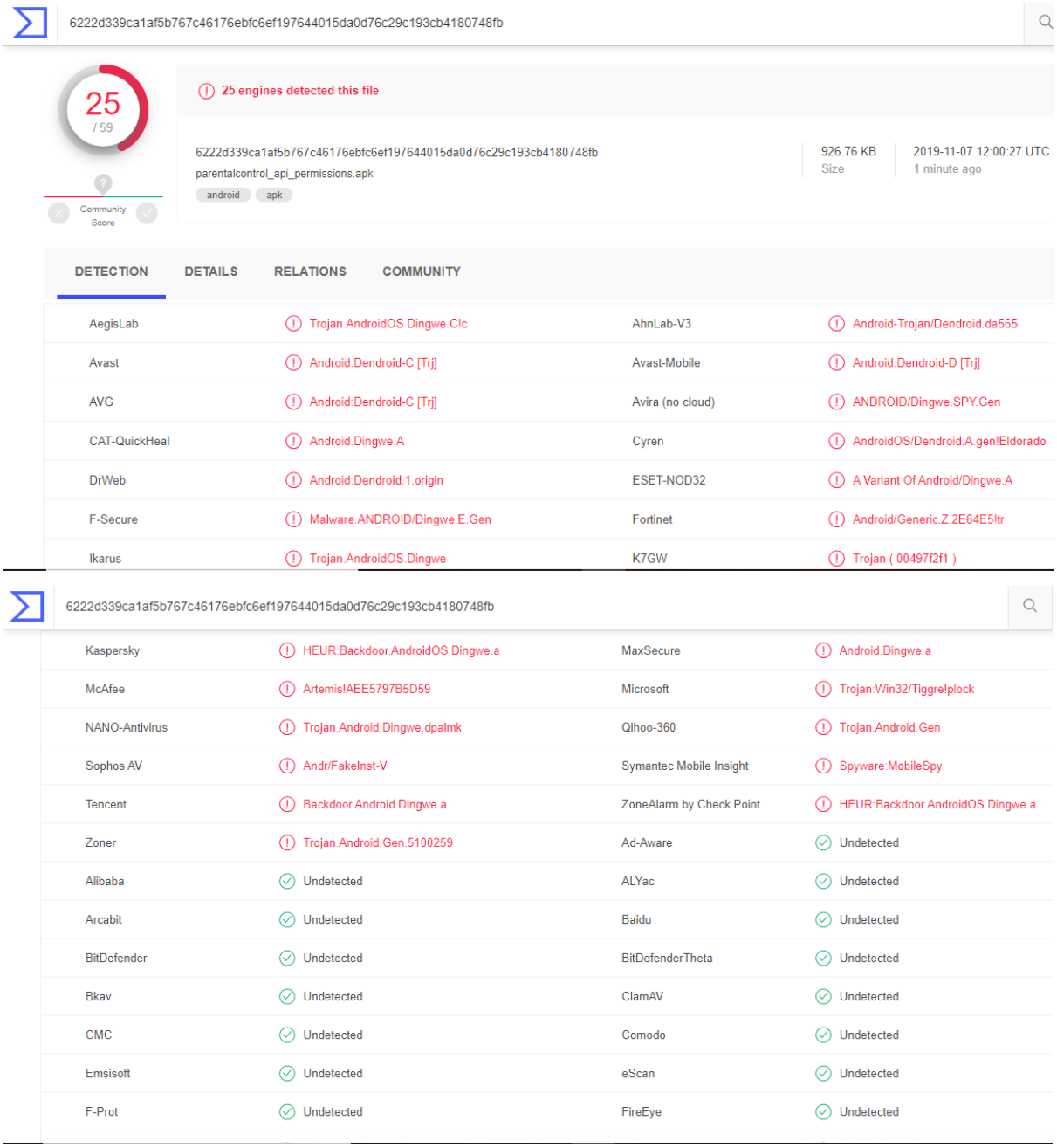



**Figure 17: VirusTotal Result for RAP Implemented Dendroid Malware**

#### 7.1.2.4.6 Insert Benign Permissions (IBP)

Inserting benign permissions produce same results as that of removing all permissions. However, it does not renders the apk non-functional. This technique also evades around 7 AMTs with 42% detection ratio and tends to evade permission-based static analysis.

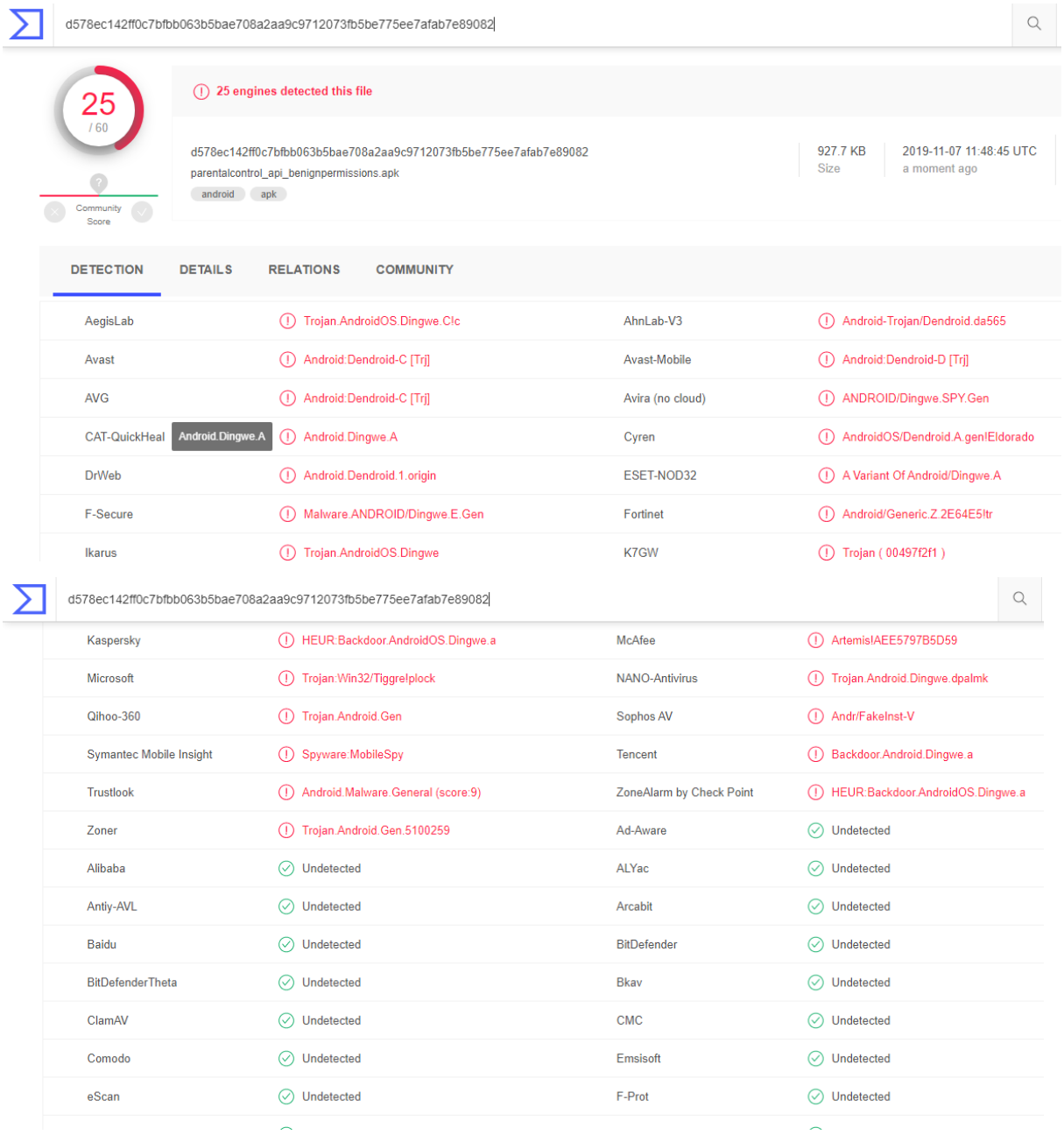



**Figure 18: VirusTotal Result for IBP Implemented Dendroid Malware**

### 7.1.2.5 Package, Class and Method Obfuscations (PCM)

PCM evasion offers 3 types of evasion capability as illustrated below:

#### 7.1.2.5.1 Change Package Name (PN_PCM)

Altering package name evades significant number of AMTs. It evades around 15 AMTs generating a detection ratio of 28.8%, approximately half of what is achieved for raw Dendroid. It must be noted that a similar subcomponent exists in API obfuscation component

which evades only 11 AMTs compared to 15 of this subcomponent. Hence, the package obfuscation subcomponent of API obfuscation must be replaced with that of PCM to achieve better results.

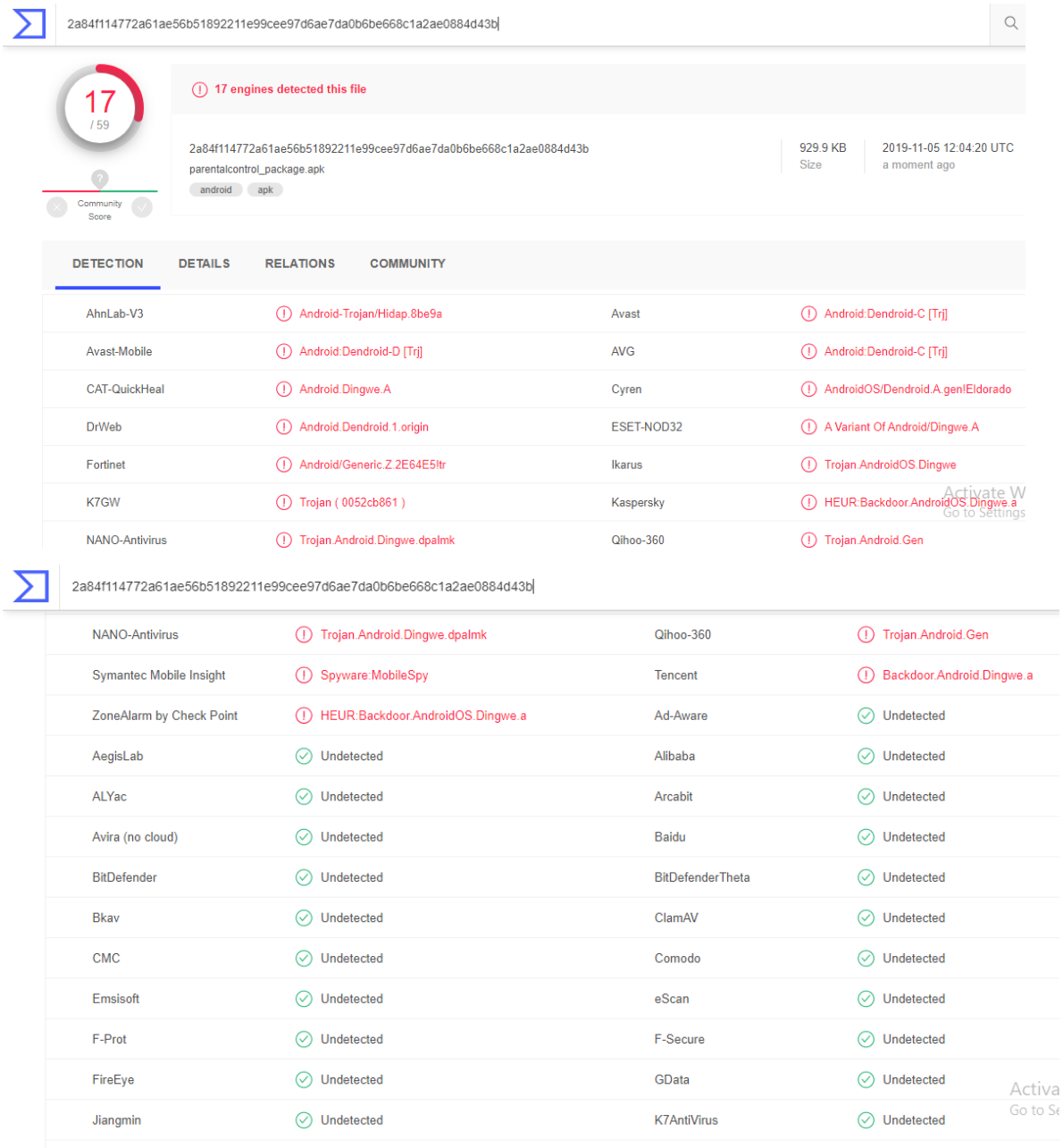



**Figure 19: VirusTotal Result for PN_PCM Implemented Dendroid Malware**

### 7.1.2.5.2 Insert Null Bytes (INB)

This subcomponent inserts null bytes between smali instructions. It evades around 14 AMTS and the detection ratio is approximately 31.5%. This evasion helps defeat signature-based detection.

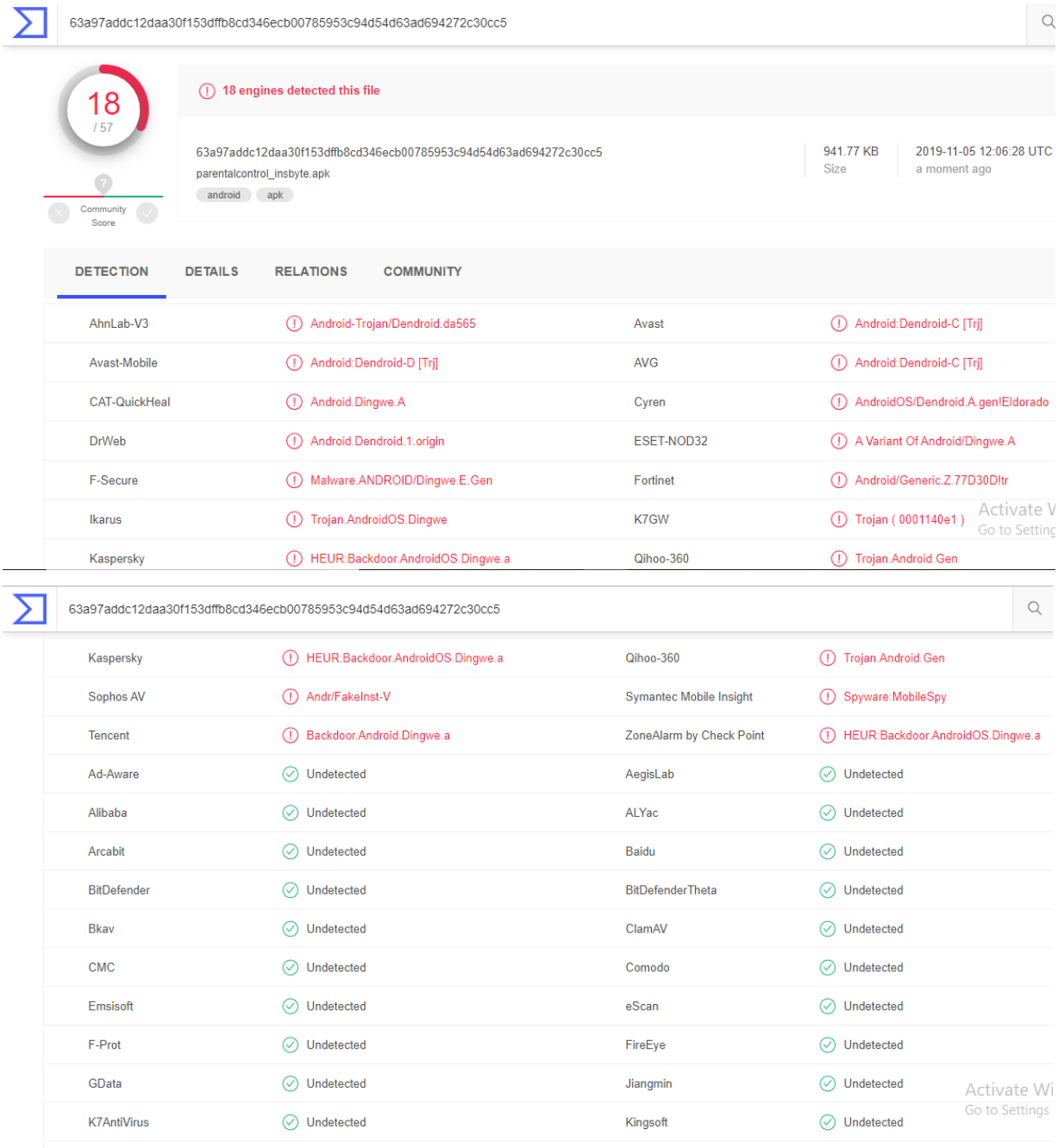



**Figure 20: VirusTotal Result for INB Implemented Dendroid Malware**

### 7.1.2.5.3 Insert Benign Class (IBC)

This method inserts benign classes into the source code to defeat the signature-based detection. It evades only 8 AMTs and detection ratio is also quite high. A 40.6% detection ratio indicates the ineffectiveness of this method.

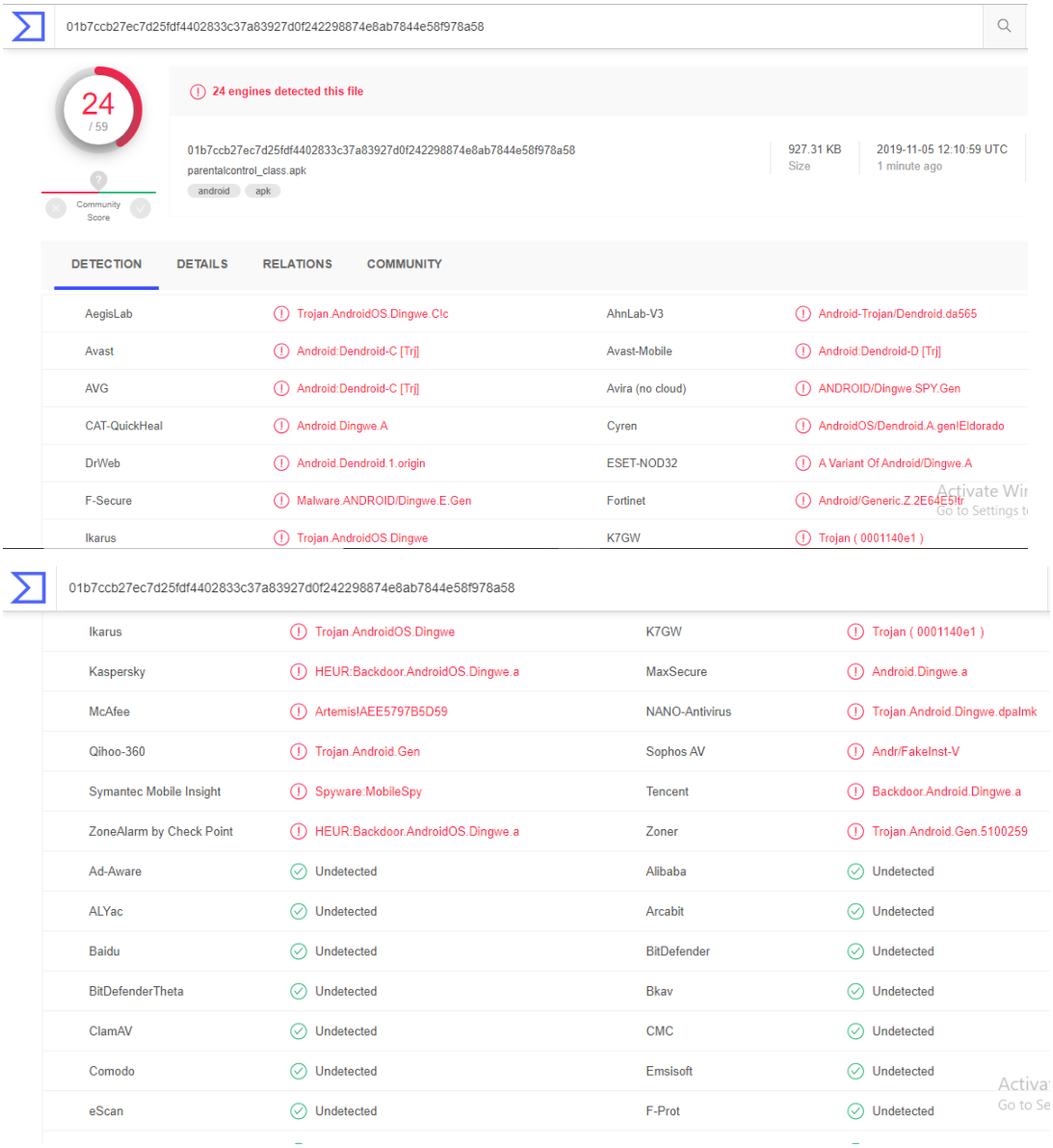



**Figure 21: VirusTotal Result for IBC Implemented Dendroid Malware**

### 7.1.2.6 Resource Obfuscations (RO)

Using resource-level obfuscations such as image modifications, xml related class references and payload nullification yield a detection ratio of 40.6% evading only 8 AMTs. These figures show that this method when implemented on its own is not much effective, however,

when implemented in conjunction with other techniques, the results are beyond effective as we will prove later.

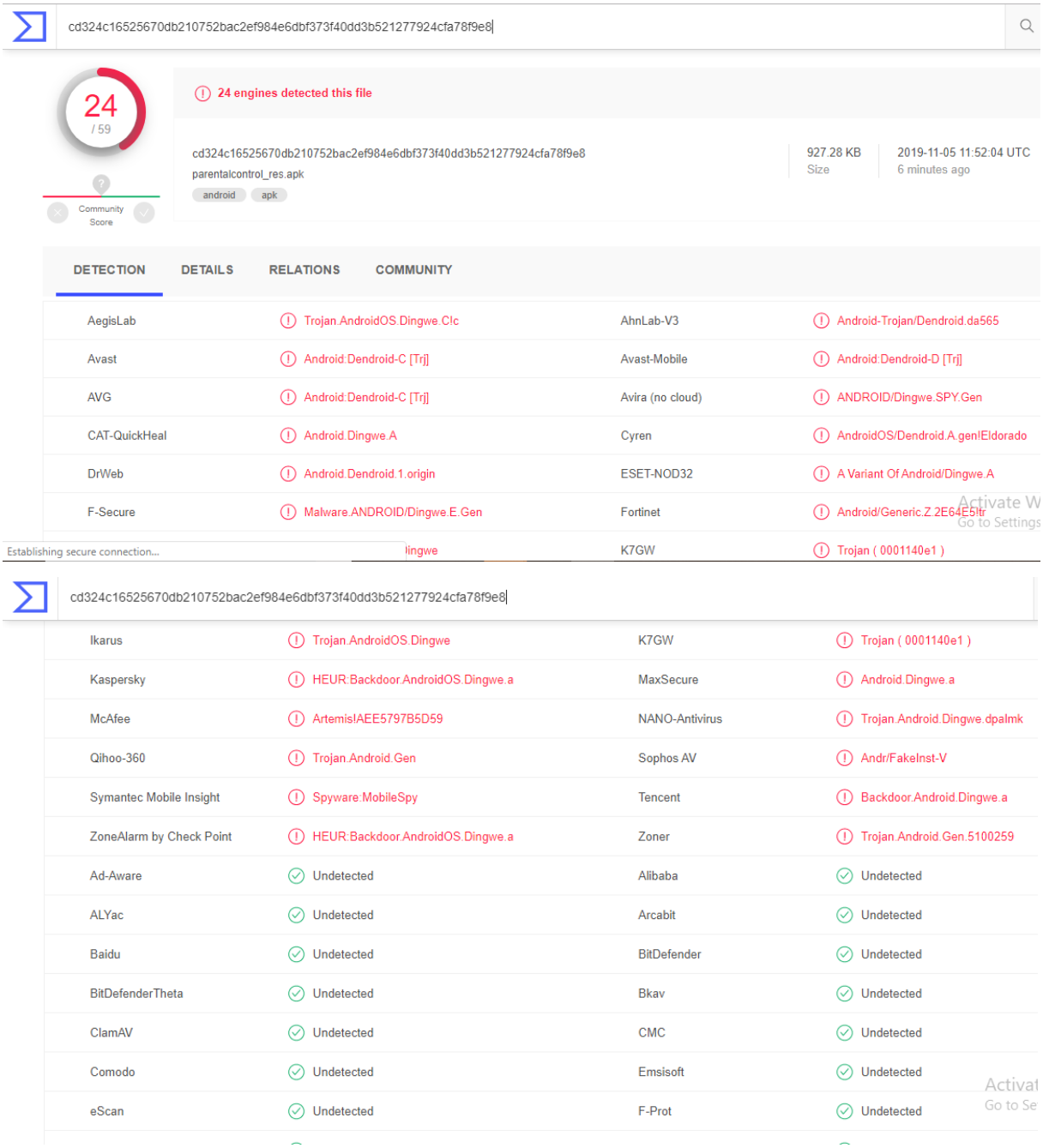



**Figure 22: VirusTotal Result for RO Implemented Dendroid Malware**

### 7.1.2.7 Angecryption (ANGE)

Employing angecryption helps evade 9 AMTs out of 32 which initially detected the raw dendroid malware. A detection ratio of approximately 40% is seen which proves that the

technique isn't much effective when implemented alone. However, when tagged along with other evasion techniques, the results are astonishing.

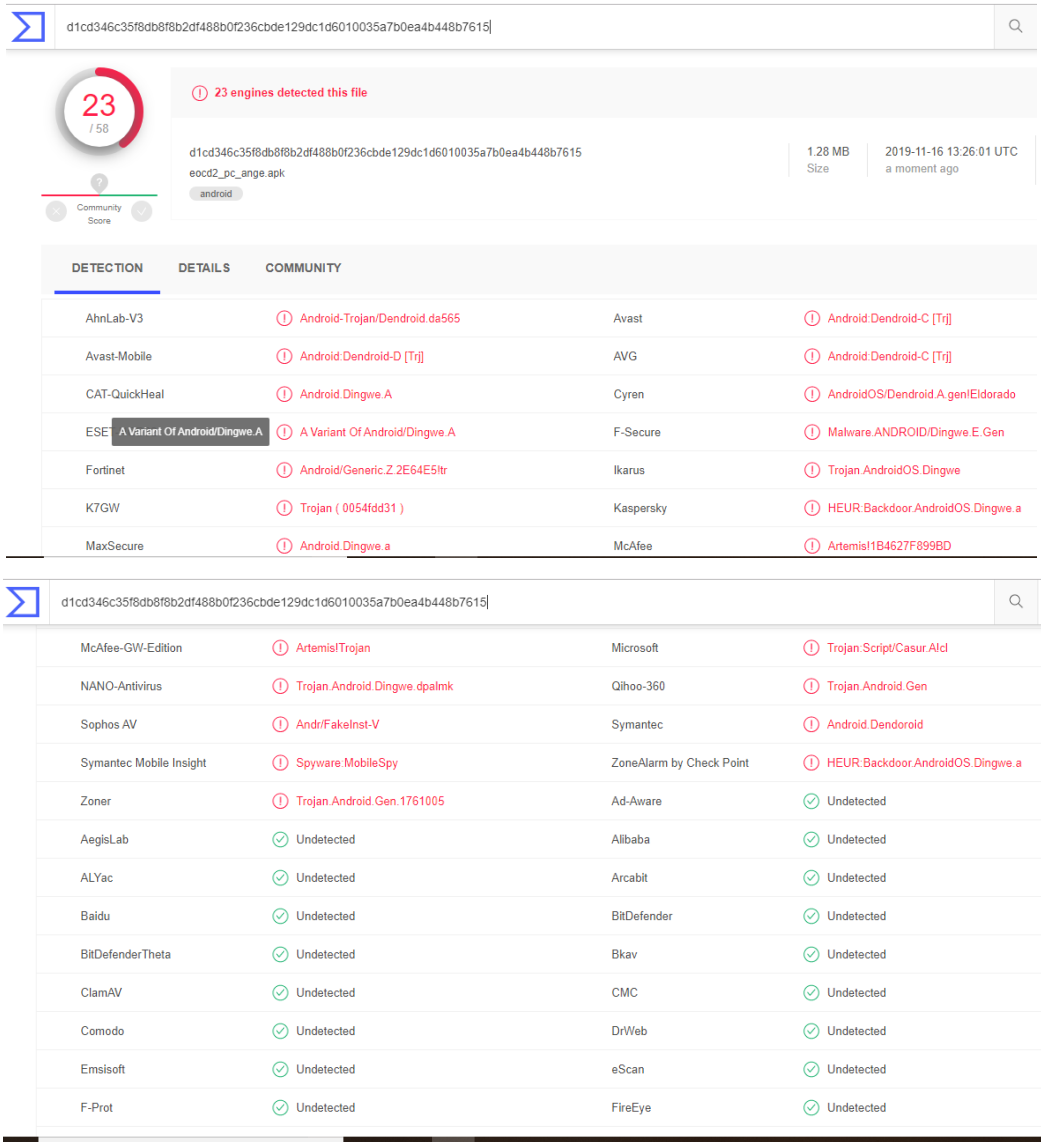



**Figure 23: VirusTotal Result for ANGE Implemented Dendroid Malware**

### 7.1.3 Multiple Evasion Module Implementation

In this phase, two or more evasion techniques are implemented together to achieve better evasion. We use a customized sequence to achieve better results as combining multiple methods evades more than one type of detection technique. Certain subcomponents are not used since they do not give good results even when used in conjunction with other such as variable encryption and all API-obfuscations as they just increase the overhead and do not

improve results. We start with string encryption and keep on adding layers of other obfuscation techniques.

#### 7.1.3.1 String Encryption (SE) + Java API Reflection (JAR)

We first apply string encryption on dendroid malware and then apply java API reflection. We now see that only 14 AMTs engines detect this file as malicious. A detection ratio of 23.7% is achieved. This evades signature-based, API-based and dataflow-based detection systems. We can see that this dual evasion module implementation improves evasion results.

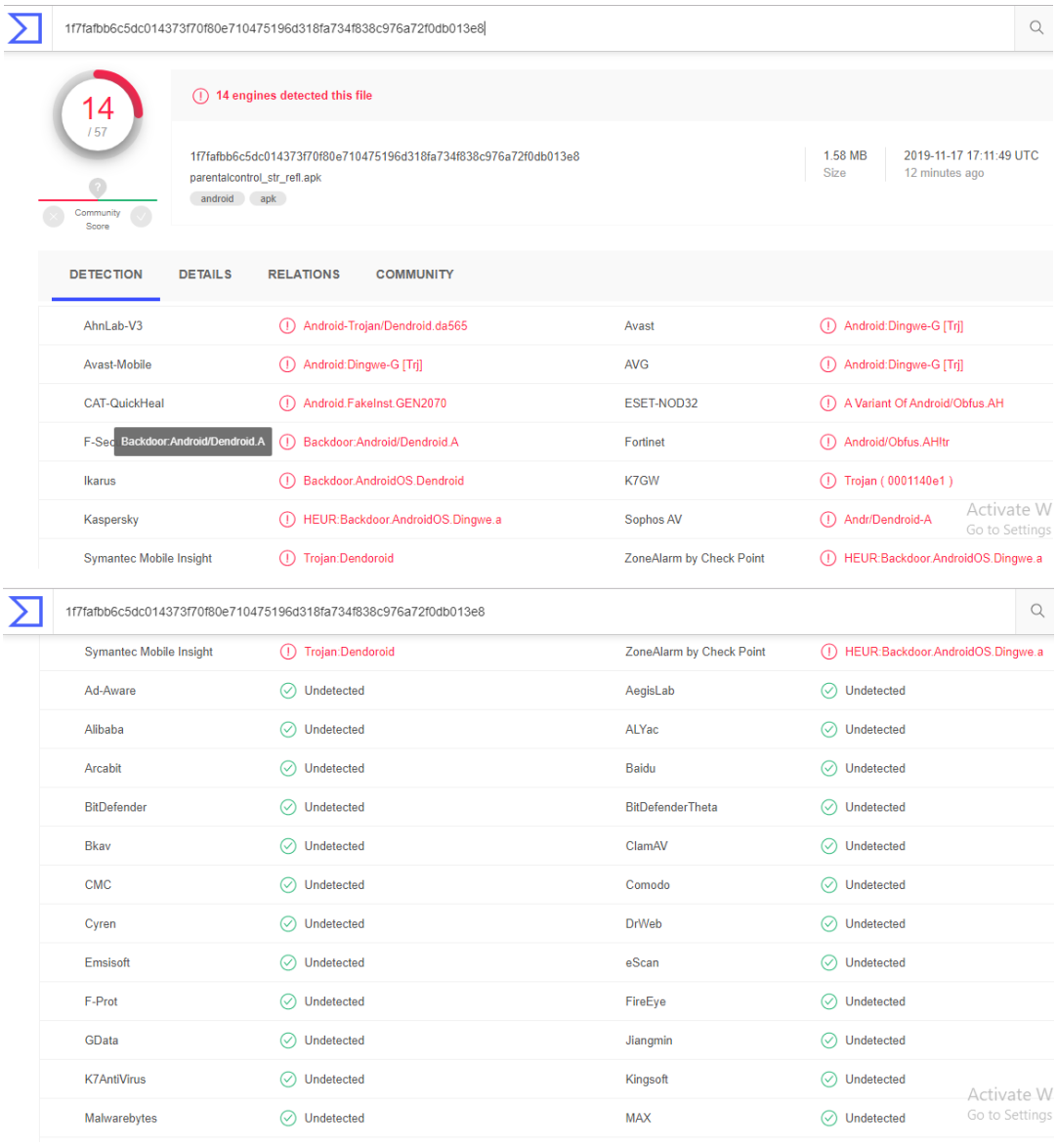



**Figure 24: VirusTotal Result for SE + JAR Implemented Dendroid Malware**

### 7.1.3.2 String Encryption (SE) + Java API Reflection (JAR) + Change Package Name (PN_PCM)

This trio of evasion method works in the sequence of the mention of its name. The result from previous dual implementation is simply put to change of package name obfuscation from PCM component. The results improve from 14 AMTs detecting to 7 AMTs detecting the Dendroid variant. The detection ratio also lowers to 11.8% from 23.7%. Thus we see that we are getting better evasion with each layer of its implementation. Using this combination, we still bypass signature-based, API-based and dataflow-based detection but with improved results.

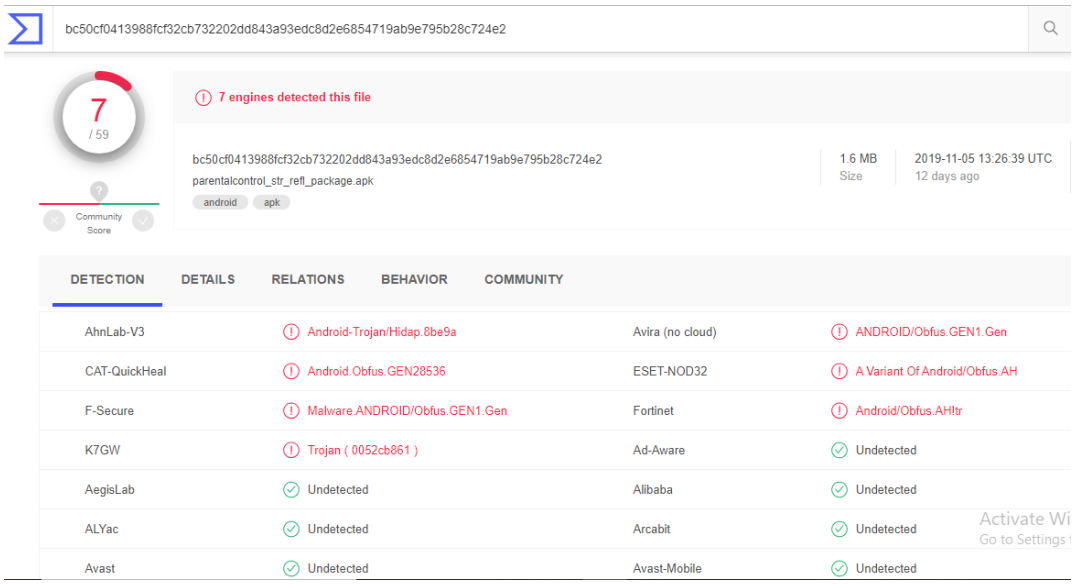



**Figure 25: VirusTotal Result for SE + JAR + PN_PCM Implemented Dendroid Malware**

### 7.1.3.3 String Encryption (SE) + Java API Reflection (JAR) + Change Package Name (PN_PCM) + Insert Null Bytes (INB)

The output apk of the previous trio is simply put to the bytecode obfuscation where nullbytes are inserted between smali instructions and resultant apk is uploaded to VirusTotal. We see that now only 5 AMTs detect Dendroid. Rest 27 are evaded and detection ratio is only 8%. Using only four obfuscation subcomponents, the degree of evasion achieved is much better than that of the standard AVPass implementation of five obfuscation subcomponents. For the standard AVPass implementation, the detection ratio was 23.7% whereas for this customized implementation, the detection ratio is only 8%.

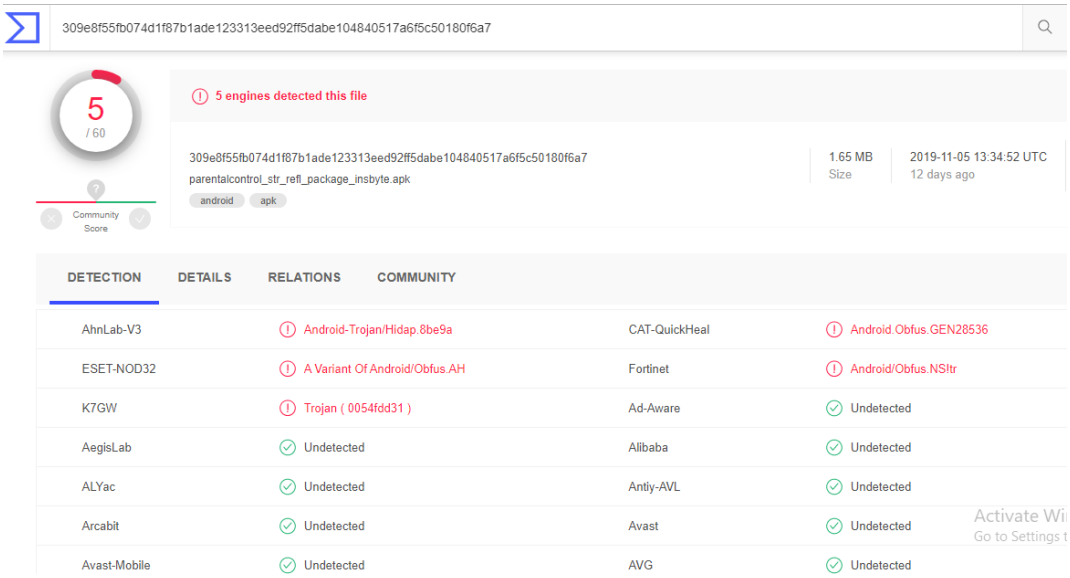



**Figure 26: VirusTotal Result for SE + JAR + PN_PCM + INB Implemented Dendroid Malware**

#### 7.1.3.4 String Encryption (SE) + Java API Reflection (JAR) + Change Package Name (PN_PCM) + Insert Null Bytes (INB) + Resource Obfuscation (RO)

We add just one module to the previous list to get better results. We reduce the number of AMTs to 2 from 32. Applying resource obfuscation gives the detection ratio of 3.4%. Only two AMTs named Ikarus and Microsoft detect the apk as malicious.

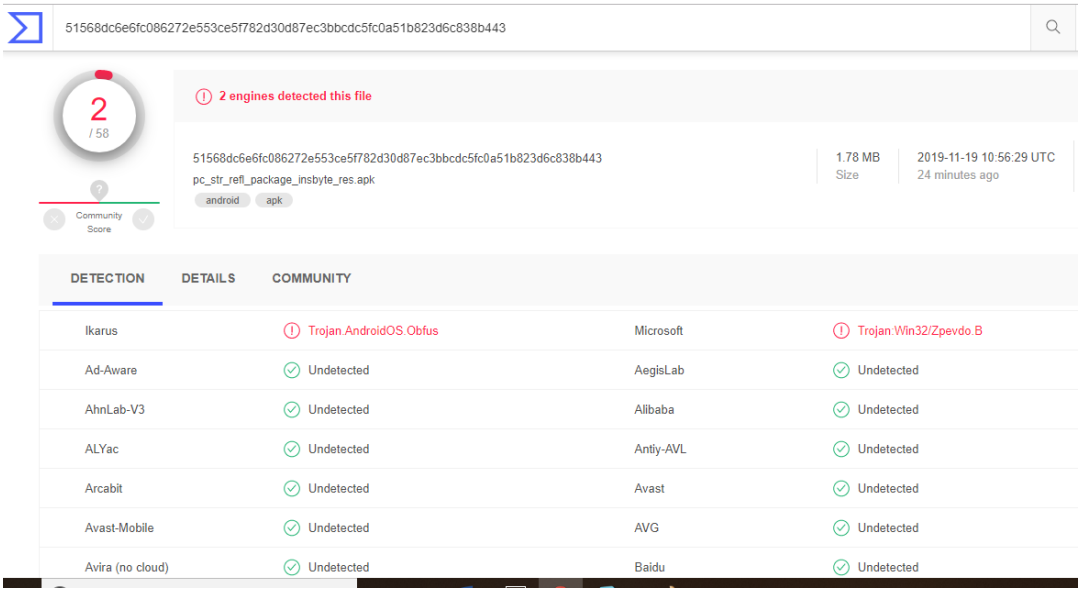



**Figure 27: VirusTotal Result for SE + JAR + PN_PCM + INB + RO Implemented Dendroid Malware**

#### 7.1.3.5 String Encryption (SE) + Java API Reflection (JAR) + Change Package Name (PN_PCM) + Insert Null Bytes (INB) + Resource Obfuscation (RO) + Angecryption (ANGE)

The resultant apk of the last step is put into the angecryption module and the outcome is an angecrypted apk. The new apk is now string encrypted, API reflected, package name altered, nullbytes inserted and angecrypted. After the application of all these subcomponents, we achieve 100% detection ratio. No AMT is able to detect the apk as malicious.

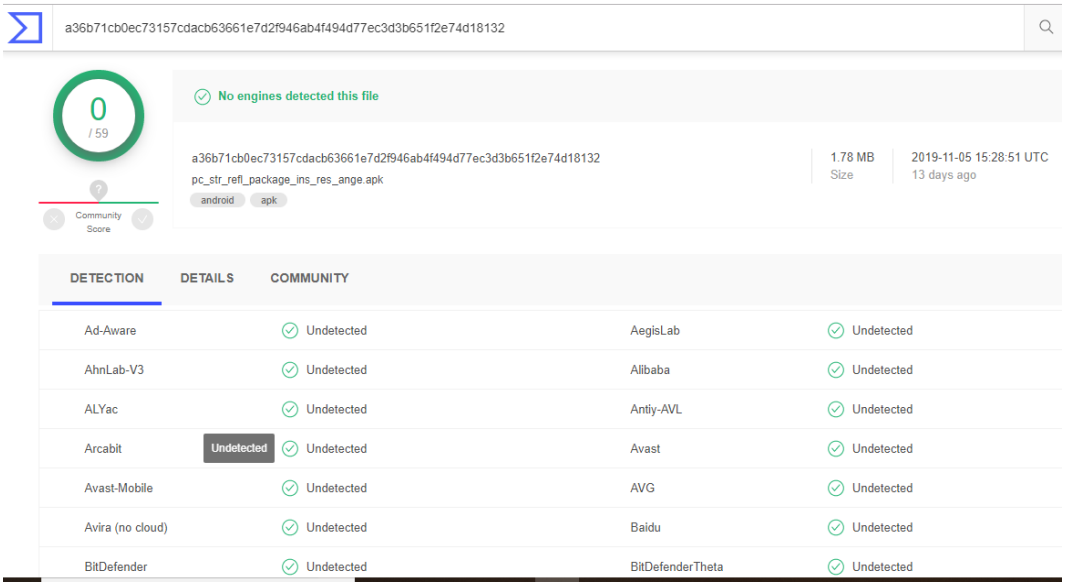



**Figure 28: VirusTotal Result for SE + JAR + PN_PCM + INB + ANGE Implemented Dendroid Malware**

#### 7.1.3.6 Java API Reflection (JAR) + String Encryption (SE) + Variable Encryption (VE) + Resource Obfuscation (RO)

AVPass when implemented in a manner as suggested by its developers yields results as depicted in the figure below. It evades around 16 AMTs among those which detects the dendroid apk as malicious. Hence a detection ratio of 23.7% is achieved. Using the customized sequence of our framework, we achieve far better results achieved than this particular method as we will prove later. AVPass first performs API reflection followed by string and variable encryption and finally modifying image and resource related xml files.

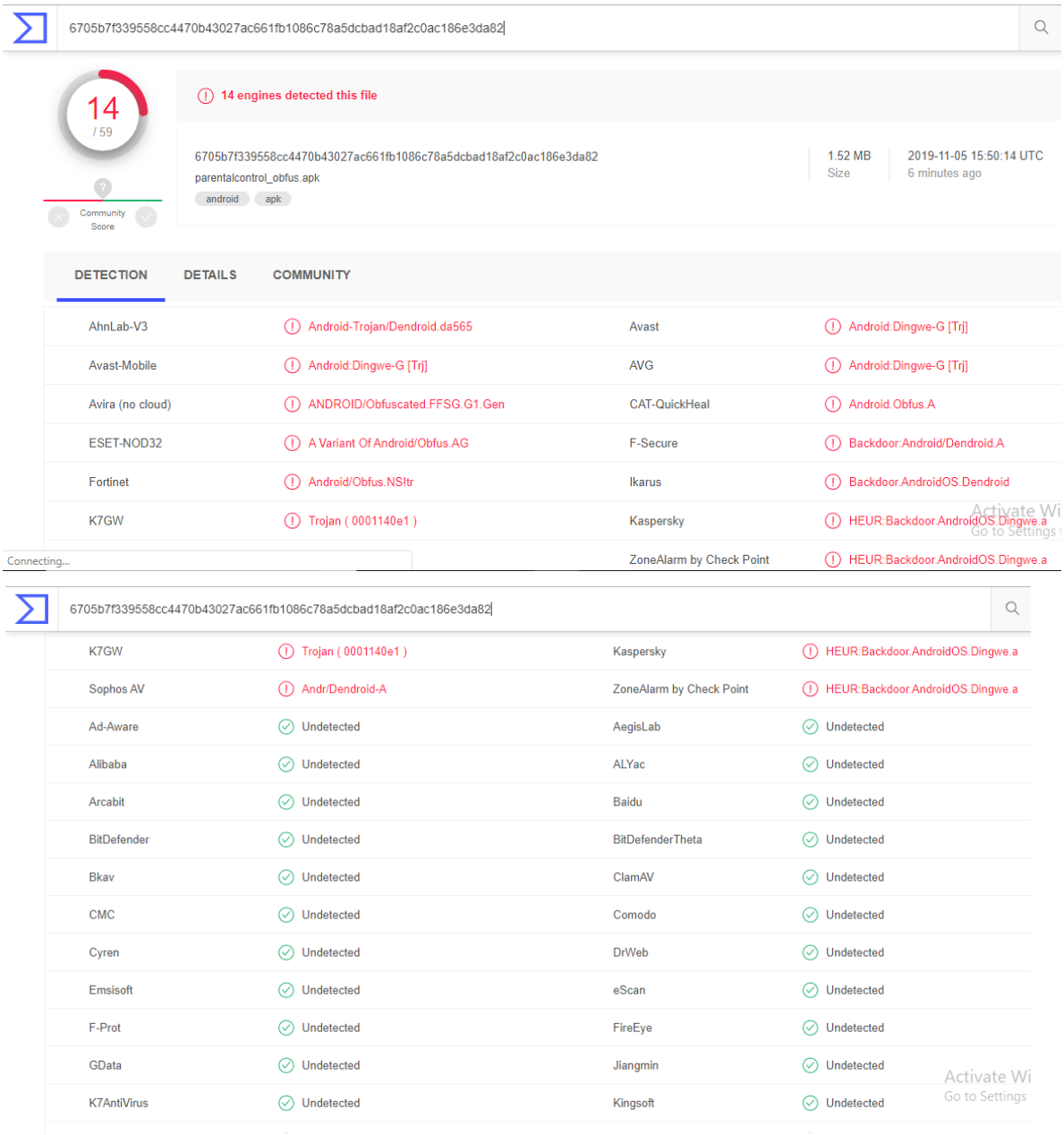



**Figure 29: VirusTotal Result for JAR + SE + VE + RO Implemented Dendroid Malware**

### 7.1.3.7 Java API Reflection (JAR) + String Encryption (SE) + Variable Encryption (VE) + Resource Obfuscation (RO) + Angecryption(ANGE)

Applying the standard AVPass obfuscation followed by angecryption helps evade 27 AMTs. The detection ratio is 8.6%. We can see that our customized implementation gives better results than the standard AVPass coupled with Angecryption implementation. We achieve 100% evasion ratio and 0% detection ratio using the customized implementation. This output

apk from AVPass+Angecryption implementation is detected by Avast, AVG, Ikarus, Kaspersky and ZoneAlarm by CheckPoint.

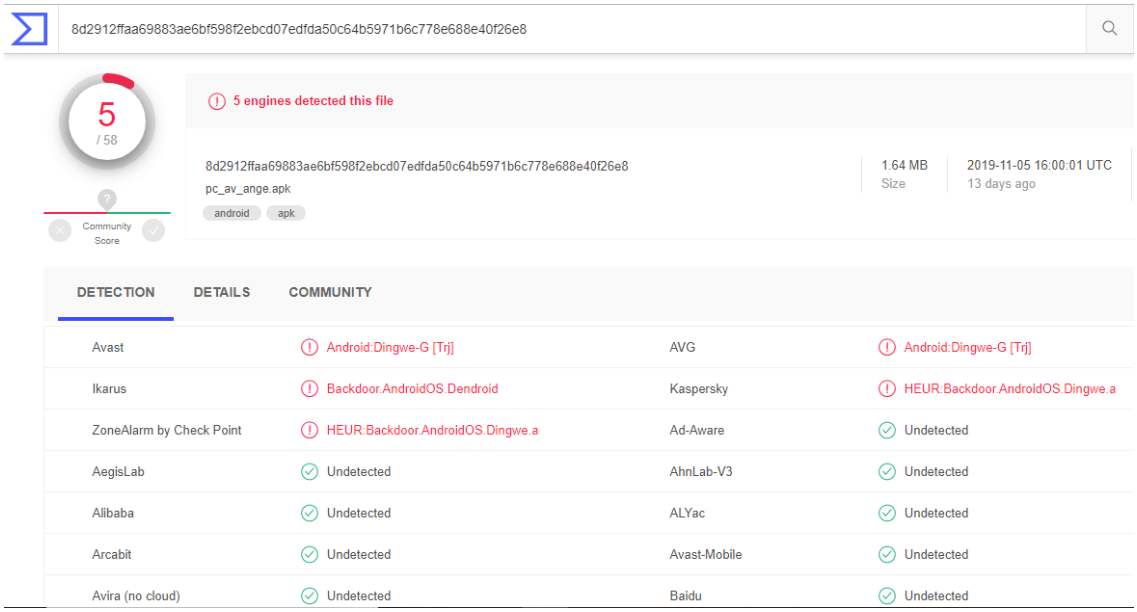



**Figure 30: VirusTotal Result for JAR + SE + VE + RO + ANGE Implemented Dendroid Malware**

## 7.2 Conclusion

We see that our solution [SE+JAR+PN_PCM+INB+RO+ANGE] gives best results among all the implementation sequences. Hence, our final output is an Android application with layers of obfuscation applied to it as demonstrated in the fig. 30 which almost camouflage the application to such an extent that no AMT is able to detect it and it appears benign, yet its malicious intent remains intact. We apply this on 200 malware samples and get approximately consistent results. Hence, we mark [SE+JAR+PN_PCM+INB+RO+ANGE] as our final evasion mechanism, however, we are flexible in implementing any evasion technique of our choice depending on the degree of evasion we want. So, our technique is not only resilient and robust but is also flexible.

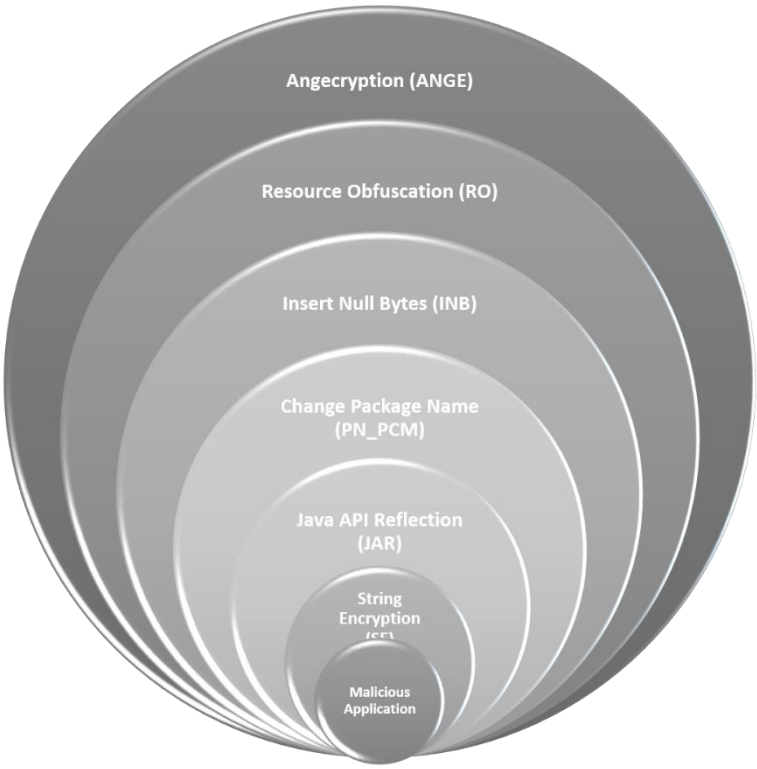



**Figure 31: Layers of Evasion Techniques Employed to the Malicious Application**

# Chapter 8

# Auditing Android Antimalware Tools (AMTs)

We applied the evasion modules as mentioned earlier iteratively, selecting best evasion components and the sequence of their application based on the detection complexity and evasion ease. In this section, we present our findings and observations.

## 8.1 Observations

When a file is uploaded to VirusTotal repository, using more than 60 AMTs, it performs a scan. The uploaded file is first hashed and stored in the database for caching purpose so as to reduce duplicate efforts and minimize the scan time. Hence, whenever a file is uploaded to VirusTotal, its hash is first looked upon in the database, upon finding a match, the results already present are displayed for that file. In case of a new file, the file's hash is calculated first, updates its hash repository, then the file sent to the AMT engines associated with VirusTotal and the returned results from these AMTs are displayed. If we only search for the new file by pasting its hash, then VirusTotal is unable to analyze the file as this new hash doesn't exists in its database. Moreover, VirusTotal is much more than AMTs aggregation. It has three Android sandboxes integrated with it apart from those for Windows such as Cuckoo for portable executables. Using Tencent HABO, Droidy and Androbox sandboxes help perform behavioral investigation of Android applications and give a meaningful insight into the working and intent of application.

## 8.2 Metrics for Auditing AMTs

The metrics used for auditing AMTs is the detection ratio obtained for each evasion technique. The higher the detection ratio, the more resilient the AMTs, the less affective the evasion technique and vice versa. The highest detection ratio is obtained for raw malware. Application of evasion techniques should lower the detection ratio and increase the number of AMTs evaded. In case of Dendroid, we set the threshold of no. of AMTs to be evaded to 32 since raw Dendroid is detected by 32 AMTs. The goal is to evade these 32 AMTs and detection ratio to be 0%.

## 8.3 Evasion of Malware Samples

The results of evasion implementation are selectively shown here in the categories as already described.

### 8.3.1 Individual Evasion Module Implementation

In this phase, individual evasion modules are implemented. As we can see both from the table and the figure, the drop in the detection ratio for single evasion module is not significant. The minimum value of detection ratio achieved is 27% for String Encryption (SE) which evades around 16 AMTs. The most easily detectable evasion techniques are those from API obfuscation, these are Remove All Permissions (RAP) and Insert Benign Permissions (IBP). The detection ratio for these two components is 42% and no. of AMTs evaded is 7 out of 32. For single evasion module's implementation, the no. of AMTs under consideration are 32 which is the no. of AMTs that detect the raw dendroid malware. The average detection ratio is 35.07% which is not very low compared that of raw malware.

**Table 7: Detection ratio and no. of AMTs evaded against individual evasion components**

| Evasion Technique | None (Raw) | SE | VE | JAR | API Obfuscation | | | | | | PCM | | | RO | ANGE |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | RP | OOP | PN_API | FN | RAP | IBP | PN_PCM | INB | IBC | | |
| Detection Ratio (%) | 53.3 | 27 | 35 | 31 | 32.2 | 32.2 | 35 | 32.2 | 42 | 42 | 28.8 | 31.5 | 40.6 | 40.6 | 40 |
| No. of AMTs evaded out of 32 | - | 16 | 12 | 14 | 13 | 13 | 11 | 13 | 7 | 7 | 15 | 14 | 8 | 8 | 9 |

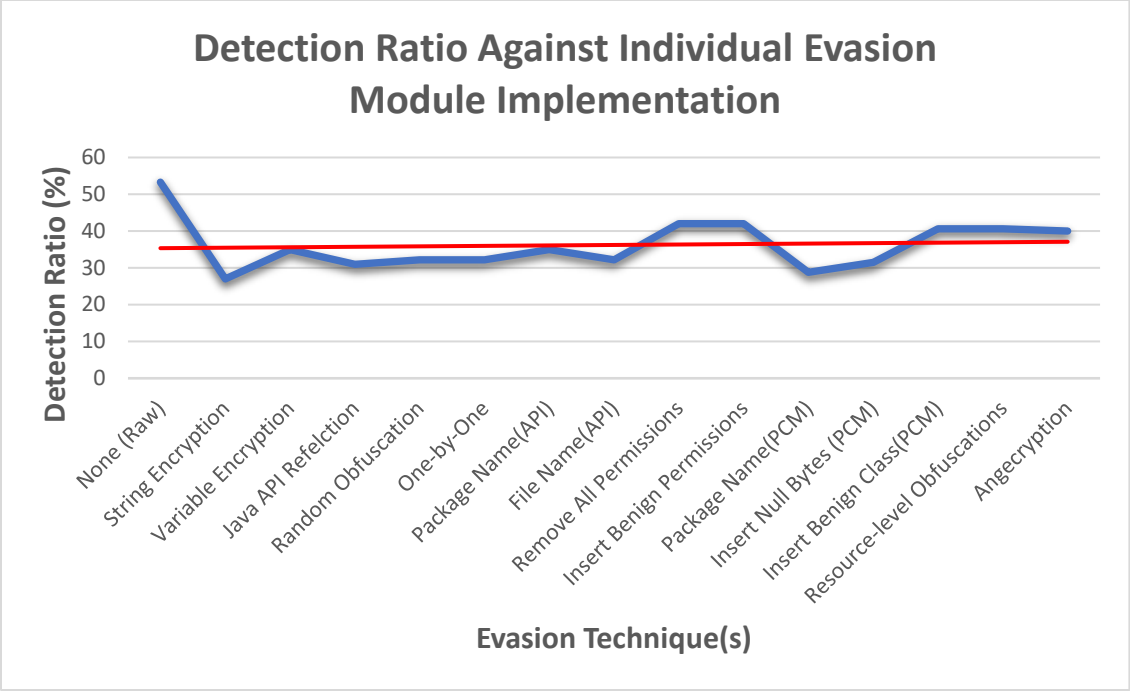



**Figure 32: Detection Ratio against Individual Evasion Implementation**

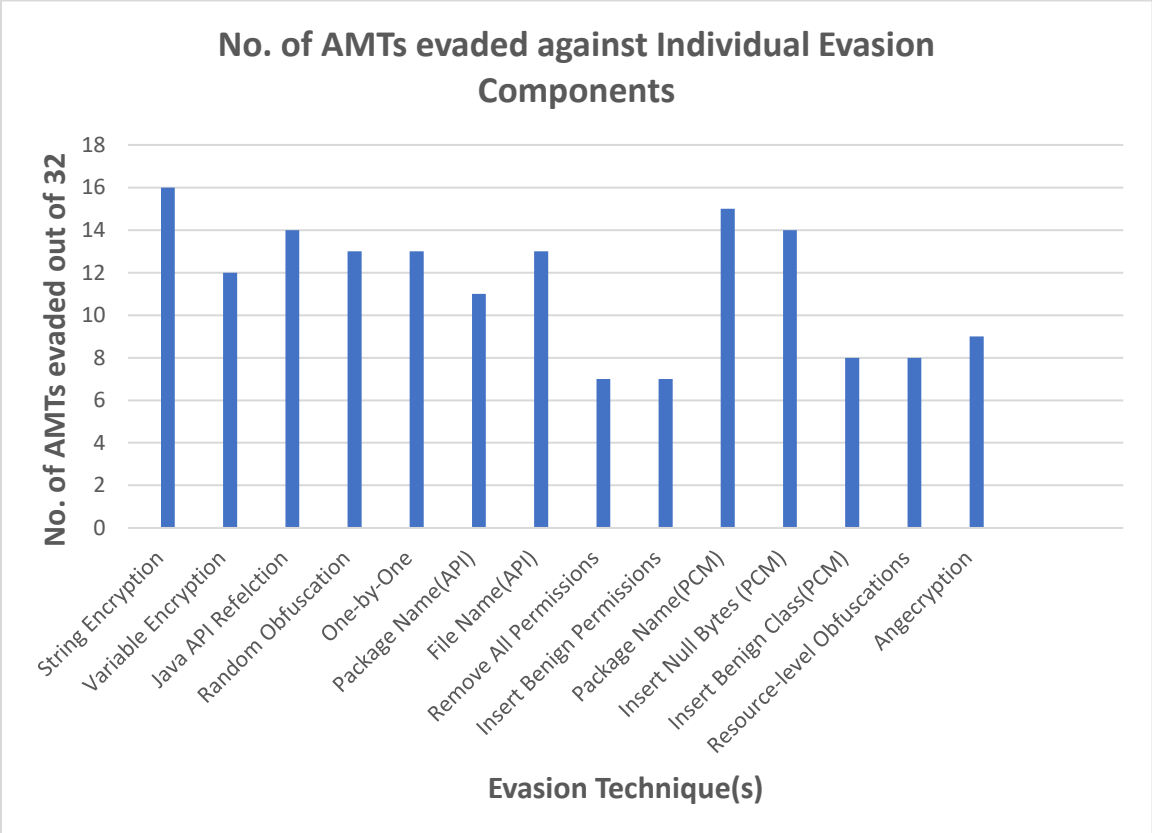



**Figure 33: No. of AMTs evaded by Individual Evasion Implementation**

Moreover, the sandboxes Droidy, Androbox and Tencent HABO integrated with VirusTotal only execute the malware in two cases, first when we perform random API perturbation and second, when one-by-one perturbation is performed. All the rest malware variants do not give off any behavioral information. The average no. of AMTs evaded is 11.42. The AMTs that do not detect any of the malware variants include Alibaba, ClamAV, Comodo, K7Antivirus, MAX and Symantec whereas the AMTs that stand out best and detect all the malware variants include AhnLab-V3, Avast, Avast-Mobile, AVG, CAT-QuickHeal, ESET-NOD32, Fortinet, K7GW and Symantec Mobile Insight. Table displays the AMTs detection capability against single evasion component's implementation. Table displays the average detection ratio and the no. of AMTs evaded against each evasion technique. This is also depicted pictorially in figures 32 and 33 respectively.

**Table 8: Single Evasion Techniques Detected by AMTs**

| Sr. No. | AMT name that detected raw malware | Evasion Techniques Detected | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | SE | VE | JAR | API Obfuscation | | | | | | PCM | | | RO | ANGE |
| | | | | | RP | OOP | PN_API | FN | RAP | IBP | PN_PCM | INB | IBC | | |
| 1. | AegisLab | × | × | × | × | × | × | × | ✓ | ✓ | × | × | ✓ | ✓ | × |
| 2. | AhnLab-V3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 3. | Alibaba | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 4. | Avast | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 5. | Avast-Mobile | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6. | AVG | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 7. | Avira | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | × | × | ✓ | ✓ | × |
| 8. | CAT-QuickHeal | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 9. | ClamAV | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 10. | Comodo | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 11. | Cyren | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 12. | DrWeb | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| 13. | ESET-NOD32 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 14. | F-Secure | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| 15. | Fortinet | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 16. | Ikarus | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ |
| 17. | K7Antivirus | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 18. | K7GW | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 19. | Kaspersky | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20. | MAX | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 21. | MaxSecure | × | × | × | × | × | × | × | ✓ | × | × | × | ✓ | ✓ | ✓ |
| 22. | McAfee | × | × | × | × | × | × | × | ✓ | ✓ | × | × | ✓ | ✓ | ✓ |
| 23. | McAfee-GW-Edition | × | × | × | × | × | × | × | × | × | × | × | × | × | ✓ |
| 24. | Microsoft | × | × | × | × | × | × | × | ✓ | ✓ | × | × | × | × | ✓ |
| 25. | NANO-Antivirus | × | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| 26. | Qihoo-360 | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 27. | Sophos AV | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| 28. | Symantec | × | × | × | × | × | × | × | × | × | × | × | × | × | × |
| 29. | Symantec Mobile Insight | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 30. | Tencent | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31. | Trustlook | × | ✓ | × | ✓ | × | ✓ | × | × | ✓ | × | × | × | × | × |
| 32. | ZoneAlarm by CheckPoint | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 33. | Zoner | × | × | × | × | × | × | × | ✓ | ✓ | × | × | × | × | ✓ |

### 8.3.2 Multiple Evasion Module Implementation

Since single evasion component were not much fruitful, hence we selected some of the best evasion components and applied them in a customized sequence. The results of the customized sequence were much better than the single components. Also, after several trials, best sequence was developed. According to [46], in order to bypass the AMTs, one needs to first apply API obfuscations followed by PCM and application of string encryption, API obfuscations, and package name better helps in detection. However, when this is implemented practically, results are not as expected. Following our implementation sequence yields best results as we have proved. Also, when we implement the AVPass in the sequence as demonstrated in the [46], only 18 AMTs are evaded out of 32. However, with our specific implementation of AVPass and Angecryption, we are able to evade all the 32 AMTs. We achieve 0% detection ratio evading all the 32 AMTs. The average detection ratio is 11.31% and the average number of AMTs evaded is 25.2.

**Table 9: Detection Ratio and No. of AMTs Evaded against Multiple Evasion Techniques**

| Evasion Technique | None (Raw) | SE+JAR | SE+JAR+PN_PCM | SE+JAR+PN_PCM+INB | SE+JAR+PN_PCM+INB+RO | SE+JAR+PN_PCM+INB+RO+ANGE | JAR+SE+VE+RO | JAR+SE+VE+RO+ANGE |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Detection Ratio (%) | 53.3 | 23.7 | 11.8 | 8 | 3.4 | 0 | 23.7 | 8.6 |
| No. of AMTs evaded out of 32 | | 18 | 25 | 27 | 30 | 32 | 18 | 27 |

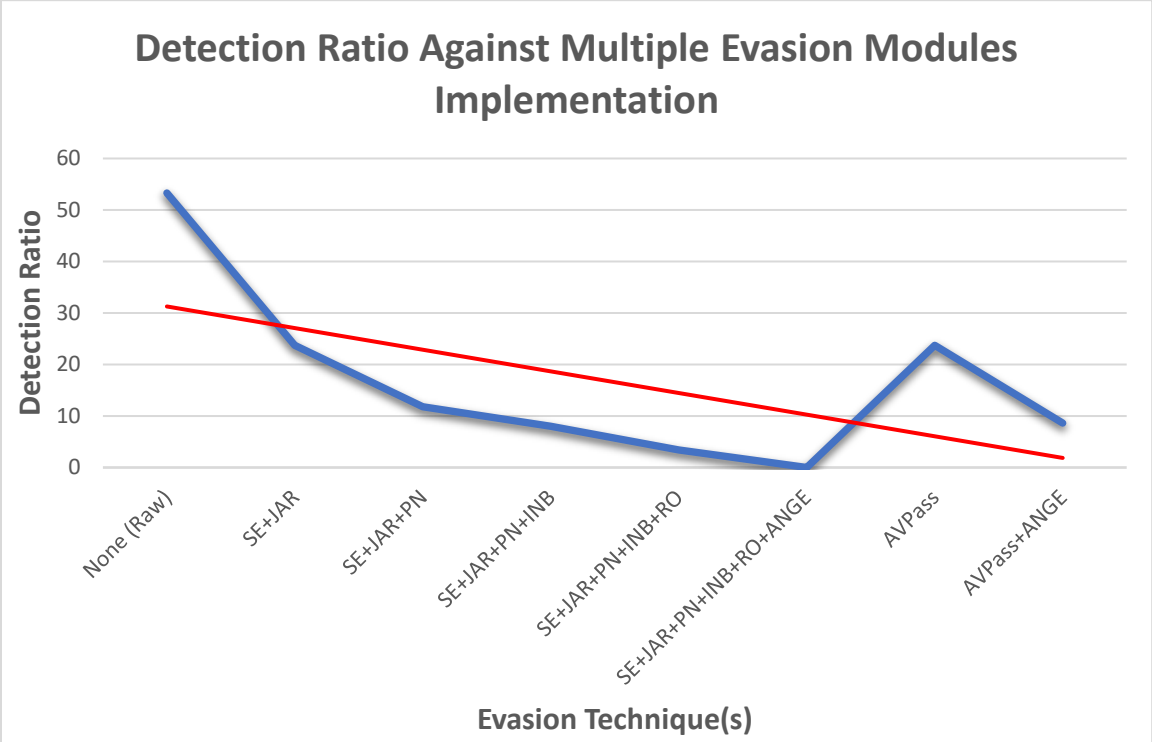



**Figure 34: Detection Ratio against Multiple Evasion Implementation**

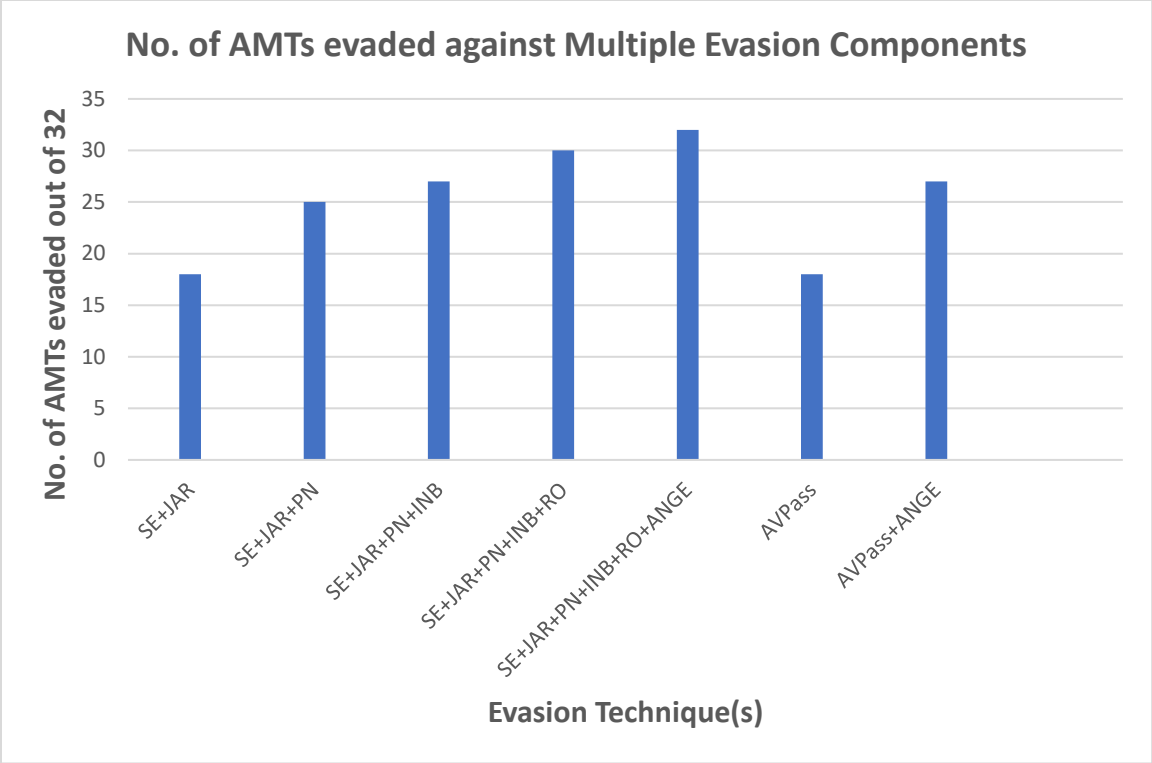



**Figure 35: No. of AMTs Evaded Against Multiple Evasion Modules Implementation**

No AMT in this case detects all the malware variants generated as a result of multiple evasion components. The least no. of detections is by Microsoft and Symantec Mobile which is only 1, followed by Sophos, Avast and Avira which are able to detect only 2 malware variants. Moreover. The maximum number of detections made by these AMTs is 4 compared to 25 of the previous phase.

**Table 10: Multiple Evasion Techniques Detected by AMTs**

| Sr. No. | AMT name | Evasion Technique | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | SE+JAR | SE + JAR + PN_PCM | SE + JAR + PN_PCM + INB | SE + JAR + PN_PCM + INB + RO | SE + JAR + PN_PCM + INB + RO + ANGE | JAR + SE + VE + RO | JAR + SE + VE + RO + ANGE |
| 1. | AhnLab-V3 | ✓ | ✓ | ✓ | × | × | ✓ | × |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2. | Avast | ✓ | × | × | × | × | ✓ | ✓ |
| 3. | Avast-Mobile | ✓ | × | × | × | × | ✓ | × |
| 4. | AVG | ✓ | × | × | × | × | ✓ | ✓ |
| 5. | Avira | × | ✓ | × | × | × | ✓ | × |
| 6. | CAT-QuickHeal | ✓ | ✓ | ✓ | × | × | ✓ | × |
| 7. | DrWeb | × | × | × | × | × | × | × |
| 8. | ESET-NOD32 | ✓ | ✓ | ✓ | × | × | ✓ | × |
| 9. | F-Secure | ✓ | ✓ | × | × | × | ✓ | × |
| 10. | Fortinet | ✓ | ✓ | ✓ | × | × | ✓ | × |
| 11. | Ikarus | ✓ | × | × | ✓ | × | ✓ | ✓ |
| 12. | K7GW | ✓ | ✓ | ✓ | × | × | ✓ | × |
| 13. | Kaspersky | ✓ | × | × | × | × | ✓ | ✓ |
| 14. | Microsoft | × | × | × | ✓ | × | × | × |
| 15. | Sophos AV | ✓ | × | × | × | × | ✓ | × |
| 16. | Symantec Mobile Insight | ✓ | × | × | × | × | × | × |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 17. | ZoneAlarm by CheckPoint | ✓ | × | × | × | × | ✓ | ✓ |

Only one evasion combination triggers behavioral investigation which is String Encryption (SE) + Java API Reflection (JAR) +Change Package Name (PN_PCM). Rest all combinations yield no behavioral information hence, running those malware variants within sandbox generates no valuable information. Thus this proves that increasing the number of evasion components, we can easily reduce the detection ratio to a minimal value. We also need to be careful about the sequence in which the evasion components are applied. Applying all the evasion components yield no better results and also renders the application no-functional. Hence, both the evasion components and their correct sequence is necessary factor in order to achieve the best results.

## 8.4 Individual AMTs

We now look at the performance of Individual AMTs. The maximum number of detections made by any AMT was 18 out of 21 times it was tested and the AMTs that earn this detection rating are AhnLab-V3, CAT-QuickHeal, ESET-NOD32, Fortinet and K7GW. These are evaded only 3 times standing resilient against the malware variants most of the time. Their performance remain consistent in both the phase of evasion implementation.

On the other hand, AMTs that could not detect any single malware variant are Alibaba, ClamAV, Comodo, K7Antivirus, MAX and Symantec. These AMTs fail to detect any obfuscated malware variant raising suspicion about their detection capability. These perform better only against un-obfuscated malware.

Moreover, AMTs that performed best in the first phase of evasion implementation and falter in the second phase include Avast, Avast-Mobile, AVG and Symantec Mobile Insight. Multiple evasion implementations abated their performance vehemently.

Ikarus and Microsoft are the only two AMTs that detect the malware at the second last step just before it fully evades all the AMTs. Their behavior is not consistent even Microsoft overall performs poorly in both the first and second phase but detects an obfuscated malware not even detectable by the best declared AMTs as mentioned earlier.

As we apply evasion techniques onto the dendroid malware, AMTs' detection signature also change. For instance, in case of AhnLab-V3, the detection signature change from Android-Trojan/Dendroid.da565 to Android-Trojan/Hidap.8be9a, for CAT-QuickHeal, signature change from Android.Dingwe.A to Android.Obfus.GEN28536 and for Ikarus, Trojan.AndroidOS.Dingwe to Trojan.AndroidOS.Obfus.

**Table 11: Signatures, No. of Detections and Evasions made by each AMTs**

| AMT name that detected raw malware | Signatures | No. of Detections | No. of times Evaded by malware variants |
|---|---|---|---|
| AegisLab | Trojan.AndroidOS.Generic.C!c | 4 | 17 |
| AhnLab-V3 | Android-Trojan/Dendroid.da565 | 18 | 3 |
| Alibaba | Backdoor:Android/Dingwe.caf18e88 | 0 | 21 |
| Avast | Android:Dendroid-C [Trj] | 17 | 4 |
| Avast-Mobile | Android:Dendroid-D [Trj] | 16 | 5 |
| AVG | Android:Dendroid-C [Trj] | 17 | 4 |
| Avira | ANDROID/Dingwe.SPY.Gen | 15 | 6 |
| CAT-QuickHeal | Android.Dingwe.A | 18 | 3 |

| | | | |
|---|---|---|---|
| ClamAV | Andr.Malware.Agent-1534052 | 0 | 21 |
| Comodo | Malware@#x1k6eaqmdedl | 0 | 21 |
| Cyren | AndroidOS/Dendroid.A.gen!Eldorado | 13 | 8 |
| DrWeb | Android.Backdoor.262.origin | 12 | 9 |
| ESET-NOD32 | A Variant Of Android/Dingwe.A | 18 | 3 |
| F-Secure | Malware.ANDROID/Dingwe.E.Gen | 16 | 5 |
| Fortinet | Android/Generic.Z.2E64E5!tr | 18 | 3 |
| Ikarus | Trojan.AndroidOS.Dingwe | 16 | 5 |
| K7Antivirus | Trojan ( 0001140e1 ) | 0 | 21 |
| K7GW | Trojan ( 0001140e1 ) | 18 | 3 |
| Kaspersky | HEUR:Backdoor.AndroidOS.Dingwe.a | 15 | 6 |
| MAX | Malware (ai Score=99) | 0 | 21 |
| MaxSecure | | 4 | 17 |
| McAfee | Artemis!DB01F96D5E66 | 5 | 16 |
| McAfee-GW-Edition | Artemis!Trojan | 1 | 20 |
| Microsoft | Trojan:Win32/Bitrep.A | 4 | 17 |
| NANO-Antivirus | Trojan.Android.Dingwe.dpalmk | 12 | 9 |

| | | | |
|---|---|---|---|
| Qihoo-360 | Trojan.Android.Gen | 13 | 8 |
| Sophos AV | Andr/FakeInst-V | 14 | 7 |
| Symantec | Trojan.Gen.2 | 0 | 21 |
| Symantec Mobile Insight | Spyware:MobileSpy | 15 | 6 |
| Tencent | Backdoor.Android.Dingwe.a | 12 | 9 |
| Trustlook | Android.Malware.General (score:9) | 4 | 17 |
| ZoneAlarm by CheckPoint | HEUR:Backdoor.AndroidOS.Dingwe.a | 16 | 5 |
| Zoner | Trojan.Android.Gen.1761005 | 3 | 18 |

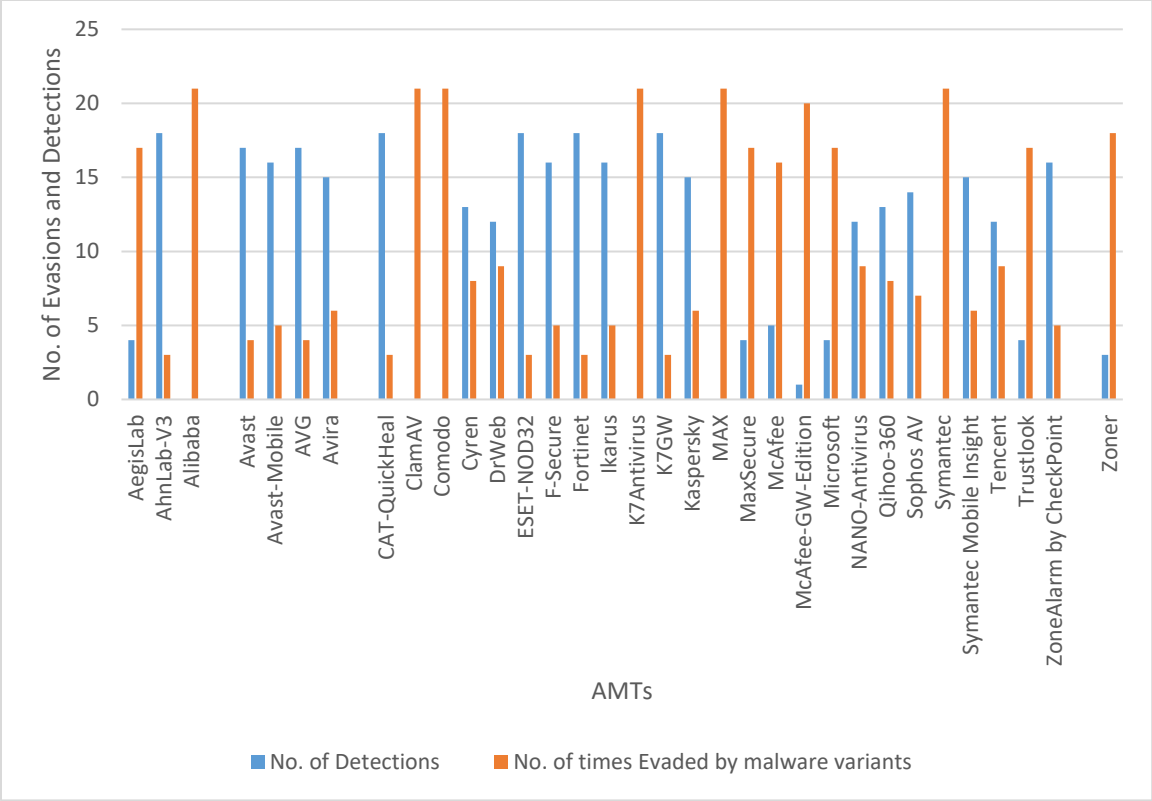



**Figure 36: No. of Evasions and Detections made by each AMT**

## 8.5 Comparison with other Techniques

We now compare our solution with some other opensource evasion techniques such as AAMO, Repacking, Angecryption, standard AVPass implementation and prove that our solution yeilds best results and better reflects the detecion efficacy of AMTs. We now present results only for the 17 best AMTs that we shortlisted in multiple evasion module implementation phase and compare the detection capability against the above listed evasion techniques. Here × represents that the AMT could not detect the malware variant and ✓ represents that it was detected by the AMT. Our solution evaded all the AMTs whereas other evasion techniques were detected most of the time. AAMO was able to evade only 4 AMTs among the given list, Repacking 2 AMTs, Angecryption only 1 AMT and standard AVPass 3 AMTs. DrWeb could not detect any of the malware variants hence its detection efficacy is worst followed by Microsoft. On the other hand, our solution was able to evade all the AMTs defeating both static analysis and fails to give any behavioral information to the sandboxes integrated with VirusTotal. Droidy, Tencent HABO and Androbox could not extract any

information about the behavior of malware. Hence both static and dynamic analysis are bypassed.

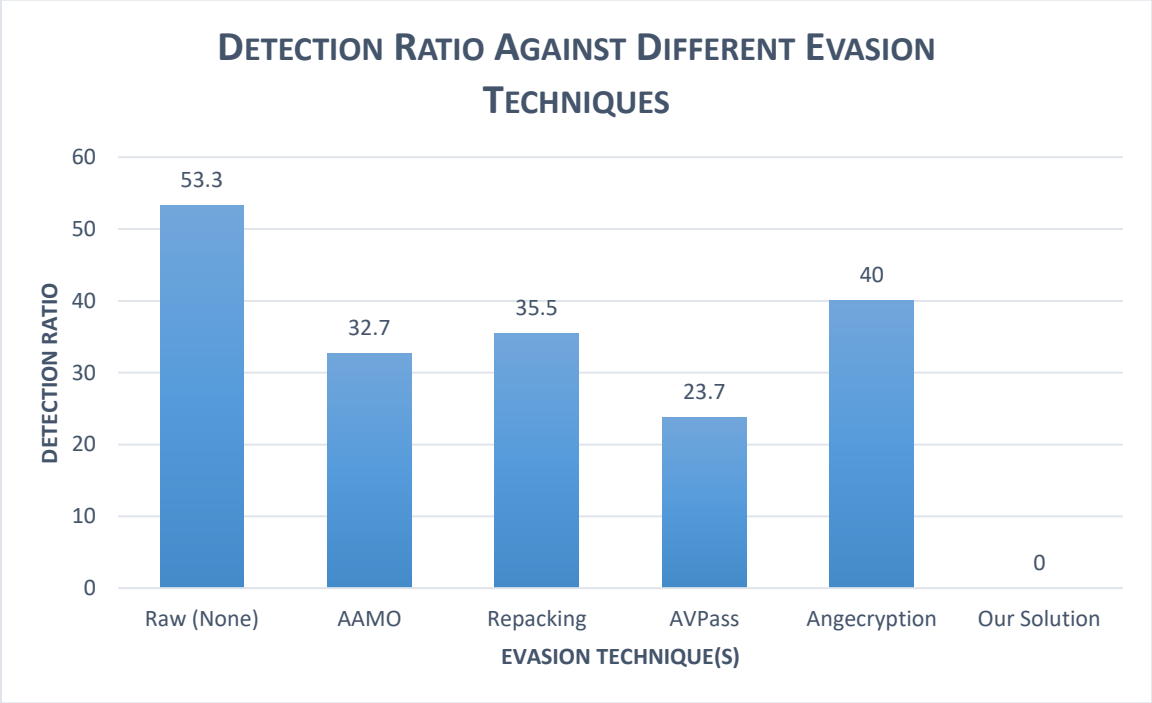



**Figure 37: Detection Ratio against Different Evasion Techniques**

The detection ratio for AAMO is 32.7% whereas that for Repacking is 35.5% which are quite high values for the detection ratio. For AVPass and Angecryption, the detection ratio are, as already mentioned, 23.7% and 40% respectively whereas our solution produces 0% detection ratio which is the desired amount when evading AMTs.

**Table 12: Comparison of Different Evasion Techniques**

| Sr. No. | Security Applications | Evasion Technique(s) Detected | | | | |
|---|---|---|---|---|---|---|
| | | AAMO | Repacking | Angecryp-tion | AVPass | Our Solution |
| 1. | AhnLab-V3 | ✓ | ✓ | ✓ | ✓ | × |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2. | Avast | ✓ | ✓ | ✓ | ✓ | × |
| 3. | Avast-Mobile | ✓ | ✓ | ✓ | ✓ | × |
| 4. | AVG | ✓ | ✓ | ✓ | ✓ | × |
| 5. | Avira | × | ✓ | ✓ | ✓ | × |
| 6. | CAT-QuickHeal | ✓ | ✓ | ✓ | ✓ | × |
| 7. | DrWeb | × | × | × | × | × |
| 8. | ESET-NOD32 | ✓ | ✓ | ✓ | ✓ | × |
| 9. | F-Secure | ✓ | ✓ | ✓ | ✓ | × |
| 10. | Fortinet | ✓ | ✓ | ✓ | ✓ | × |
| 11. | Ikarus | × | ✓ | ✓ | ✓ | × |

| | | | | | | |
|---|---|---|---|---|---|---|
| 12. | K7GW | ✓ | ✓ | ✓ | ✓ | × |
| 13. | Kaspersky | ✓ | ✓ | ✓ | ✓ | × |
| 14. | Microsoft | × | × | ✓ | × | × |
| 15. | Sophos AV | ✓ | ✓ | ✓ | ✓ | × |
| 16. | Symantec Mobile Insight | ✓ | ✓ | ✓ | × | × |
| 17. | ZoneAlarm by CheckPoint | ✓ | ✓ | ✓ | ✓ | × |

# Chapter 9

# Conclusion and Future Work

No antimalware solution detects the evasive malicious application and could not process the obfuscated malware. Hence this raises questions about the detection efficacy of these antimalware solutions which rely on conventional detection techniques as mentioned earlier and depicts a huge gap in the Android Antimalware domain. For developing a robust and resilient malware detector for Android, there is a need to first identify shortcomings and flaws in the current detection systems and overcoming those flaws in the new AMTs.

There is a need to adopt a hybrid policy comprising of both static and dynamic analysis techniques and integrating modules to scan not only code and runtime analysis but also conduct a deep scan dissecting every section of the application under observation. Antimalware engines must be able to first detect the presence of obfuscated malware and detect all of its possible variants.

## 9.1 Conclusion

With mobile device's inadequate processing capacity, standalone AMTs for Android must be resilient enough to detect both known and variants of known malware based on signatures. Using different evasion techniques, the effectiveness of malware detectors was put under test and in most cases malware detectors performed below par the expectations.

Selecting both appropriate evasion modules and their apposite sequence is very critical in evading maximum number of AMTs. We divided our evasion implementation task into phases, first testing single components then testing combination of several components and after selection of the correct components and their implementation sequence, we audit the several state-of-the-art AMTs. We found that some evasion modules when implemented as a single evasion components better bypass the AMTs as compared to others such as String Encryption (SE) component and Changing Package Name (PN_PCM) component evading more than 15 AMTs while some others perform poorly such as some components of API obfuscation and resource-level obfuscations evading as low as 7 AMTs out of 32. On the

other hand, implementing these evasion components in a certain sequence iteratively yields better results compared to some other sequences as we demonstrated in the previous chapter. Changing package name (PN_PCM) was one of the evasion component which in conjunction with SE and JAR greatly reduced the number of detections from 14 to 7. Hence, more the number of evasion components used, the lower is the number of detectors that detect the malware.

We inferred that AMTs detect chiefly by signature matching. Some instances of interaction-based and dataflow-based detection was found when behavioral investigation was performed on malware samples where we noticed network communication, http requests, files written and deleted by the application, services started by the malware etc. However, this behavioral investigation was only performed in 3 cases out of 21 evasion techniques implemented. In other cases, no behavioral information was presented by VirusTotal rather only basic details such as permissions, application's format information etc. was available.

## 9.2 Future Work

It must be noted that the evasion techniques used in the proposed solution were based on obfuscation and encryption. Hence, in order for AMTs to detect such malware variants, there should be some mechanism to de-obfuscate, and decrypt the contents of such Android malwares. If employing de-obfuscation and decryption in detecting malware variants turns out successful, we can develop de-obfuscation and decryption algorithms and integrate in AMTs. As mentioned earlier, the evasion achieved is greater when we use certain evasion component thus proving that AMTs put more focus on certain aspects of the Android applications and much less on others. Hence Android Antimalware engines must take into account this factor in their detection algorithm.

Most of the Antimalware engines do not reveal any significant information about their detection mechanism. If we have access to AMTs source code and working, we can integrate defense mechanism against such obfuscation techniques within these AMTs.

With this evasion mechanism in place, we can try different malware datasets and dynamic analysis tools, infer new detection features if these tools detect these evasion techniques. We can aggregate these detection features into repository of hybrid malware features collected

both from static and dynamic analysis. Based on this repository, we can develop malware detectors that use low processing power, handy for the devices such as Android and use the features present on the hybrid malware features repository. This will help them perform dynamic analysis in addition to static analysis and dissect the malware variants to the core to find any hidden intent of the applications.

One other factor that needs to be considered is that AMTs for Android don't have root access whereas on Windows, AVs have privileged access due to which their performance is exclusive and topnotch. Some malware variants reveal their malicious intent only when given root access. If AMTs don't have root access, they cannot detect such malware variants as such malware use the lack of this feature into their advantage and do not reveal their malicious intent, hence marked benign by the anti-malware engines. This factor hinders the detection capability of Antimalware engines for Android. AMTs on Android platform must have privileged access so that they can provide better defense against malicious applications.

# Bibliography


[1] N. Elenkov, Android security internals. San Francisco, CA: No Starch Press, 2015.

[2] Android Hacker's Handbook by Joshua J. Drake, Pau Oliva Fora, Zach Lanier, Collin Mulliner, Stephen A. Ridley, Georg Wicherski, Published by John Wiley & Sons, Inc. 10475 Crosspoint Boulevard Indianapolis, IN 46256 www.wiley.com

[3] 2019 [Online]. Available: https://www.malwarebytes.com/android-antivirus/. [Accessed: 08- Mar- 2019]

[4] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," in Symp. on Usable Privacy and Security. Washington: Advancing Science, Serving Society, March 2012

[5] J. Fenske, "Biometrics in new era of mobile access control," Biometric Technology Today, vol. 2012, no. 9, pp. 9–11, 2012.

[6] N. Husted, H. Sa¨ıdi, and A. Gehani, "Smartphone security limitations: conflicting traditions," in Proc. 2011 Workshop on Governance of Technology, Information, and Policies, ser. GTIP '11. New York, NY, USA: ACM, 2011, pp. 5–12

[7] Suarez-Tangil, Guillermo & Tapiador, Juan & Peris-Lopez, Pedro & Ribagorda, Arturo. (2013). Evolution, Detection and Analysis of Malware for Smart Devices. IEEE Communications Surveys &amp Tutorials. 16. 10.1109/SURV.2013.101613.00077.

[8] "KSB_statistics_2018_eng_final.pdf", Go.kaspersky.com, 2019. [Online]. Available: https://go.kaspersky.com/rs/802-IJN-240/images/KSB_statistics_2018_eng_final.pdf. [Accessed: 09- Mar- 2019]

[9] Symantec.com, 2019. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/eports/istr-23-2018-en.pdf.

[10] Mcafee.com, 2019. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf. [Accessed: 09- Mar- 2019]

[11] "Android malware detection evasion and resilience techniques: some examples | So Long, and Thanks for All the Fish", So Long, and Thanks for All the Fish, 2019. [Online]. Available: https://www.andreafortuna.org/technology/android/android-malware-detection-evasion-and-resilience-techniques-some-examples/.

[12] J. Marpaung, M. Sain and H. Lee, "Survey on malware evasion techniques: State of the art and challenges", in 2012 14th International Conference on Advanced Communication Technology (ICACT), PyeongChang, South Korea, 2012.

[13] S. Badhani and S. Muttoo, "Evading android anti-malware by hiding malicious application inside images", International Journal of System Assurance Engineering and Management, vol. 9, no. 2, pp. 482-493, 2017.

[14] "Android Malware Genome Project", Malgenomeproject.org, 2019. [Online]. Available: http://www.malgenomeproject.org/.

[15] D. Arp, "The Drebin Dataset", Sec.cs.tu-bs.de, 2019. [Online]. Available: https://www.sec.cs.tu-bs.de/~danarp/drebin/.

[16] "Android Open Source Project", Android Open Source Project, 2019. [Online]. Available: https://source.android.com/security. [Accessed: 10- July- 2019]

[17] Umasankar, "Analysis of latest vulnerabilities in android," 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Udupi, 2017, pp. 1236-1241. doi: 10.1109/ICACCI.2017.8126011

[18] Oracle, JAR File Specification, http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html

[19] SELinux Project, SE for Android, http://selinuxproject.org/page/SEAndroid Linux kernel source tree, *dm-verity*, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linuxgit/tree/Documentation/device-mapper/verity.txt

[20] "Android Security Bulletins | Android Open Source Project", Android Open Source Project, 2019. [Online]. Available: https:// source.android.com/security/bulletin. [Accessed: 10- Mar- 2019]

[21] "Information Disclosure", Docs.microsoft.com, 2019. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/ wcf/feature-details/information-disclosure. [Accessed:10-Mar- 19]

[22] Android Vulnerability NVD Results: https://nvd.nist.gov/vuln/search/results?adv_search=false&form_type=basic&results_type=overview&search_type=all&query=android

[23] “Android and Security - Official Google Mobile Blog.” [Online]. Available:https://www.blog.google/topics/safety-security/shielding-you-potentially-harmful-applications/ html.

[24] “Android and Security - Official Google Mobile Blog.” [Online]. Available: https://www.blog.google/topics/safety-security/shielding-you-potentially-harmful-applications/ html.

[25] R. Raveendranath, V. Rajamani, A. J. Babu, and S. K. Datta, “Android malware attacks and countermeasures: Current and future directions,” 2014 Int. Conf. Control. Instrumentation, Co “the apple threat landscape”Symantec, [online]. Available:

[26] http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/applethreat-landscape.pdf. mmun. Comput. Technol., pp. 137–143, 2014.

[27] “root exploits.” [Online]. Available: http://www.selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid. pdf.

[28] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets,” Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp., no. 2, pp. 5–8, 2012

[29] New Threats and Countermeasures in Digital Crime and Cyber Terrorism. IGI Global, 2015.

[30] “Android.Fakedefender.B | Symantec.” [Online]. Available: https://www.symantec.com/security_response/writeup.jsp?docid=2013- 091013-3953-99.

[31] Ransomware scammers exploited Safari bug to extort porn-viewing iOS users". Available at : https://arstechnica.com/information technology/2017/03/ransomware - scammers-exploited-safaribug-to-extort-porn-viewing-ios-users/.

[32] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” 2012 IEEE Symp. Secur. Priv., no. 4, pp. 95–109, 2012.

[33] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” 2012 IEEE Symp. Secur. Priv., no. 4, pp. 95–109, 2012.

[34] C. a Castillo, “Android Malware Past , Present , and Future,” McAfee White Pap. Mob. Secur. Work. Gr., pp. 1–28, 2011

[35] Amro, Belal. (2017). Malware Detection Techniques for Mobile Devices. nternational Journal of Mobile Network Communications & Telematics. 7. 10.5121/ijmnct.2017.7601.

[36] “A Look at Repackaged Apps and their Effect on the Mobile Threat Landscape.” [Online]. Available: http://blog.trendmicro.com/trendlabs- security-intelligence/a-look-into-repackaged-apps-and-its-rolein-the- mobile-threat-landscape/.

[37] “NotCompatible Android Trojan: What You Need to Know | PCWorld.” [Online]. Available: http://www.pcworld.com/article/254918/notcompatible_android_trojan_what_you_need_to_know.html.

[38] G. Nellaivadivelu, "Black Box Analysis of Android Malware Detectors," San Jose State University, San Jose, 2017.

[39] 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

[40] 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

[41] [51] [Cybrary. (2019). Hide Secret Message Inside an Image Using LSB-Steganography - Cybrary. [online] Available at: https://www.cybrary.it/0p3n/hide-secret-message-inside-image-using-lsb-steganography/ [Accessed 11 Mar. 2019].

[42] M. 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, vol. 13, no. 3, pp. 209-232, 2016.

[43] "Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps.", Ibotpeaches.github.io, 2019. [Online]. Available: https://ibotpeaches.github.io/Apktool/. [Accessed: 10- Mar- 2019]

[44] "http://www.oracle.com/," [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html. [Accessed 29 November 2019]. [1]

[45] "https://developer.android.com/," [Online]. Available: https://developer.android.com/studio/command-line/zipalign. [Accessed 29 November 2019].

[46] C. J. M. W. Y. a. T. K. Jinho Jung, "AVPASS: Automatically Bypassing Android Malware Detection System," Mandalay Bay/ Las Vegas, 2017.

[47] "http://www.skyfree.org/," [Online]. Available: http://www.skyfree.org/linux/references/ELF_Format.pdf. [Accessed 29 November 2019].

[48] "https://developer.android.com/," [Online]. Available: https://developer.android.com/reference/dalvik/system/DexClassLoader. [Accessed 29 November 2019].

[49] Y. C. X. J. Vaibhav Rastogi, "Evaluating Android Anti-malware against Transformation Attacks," Northwestern University, North Carolina State University, North Carolina, 2013.

[50] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang and T. Chen, "Mystique: Evolving Android Malware for Auditing Anti-Malware Tools", 2019.

[51] Y. Xue, G. Meng, Y. Liu, T. Tan, H. Chen, J. Sun and J. Zhang, "Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique", IEEE Transactions on Information Forensics and Security, vol. 12, no. 7, pp. 1529-1544, 2017.

[52] S. Sen, E. Aydogan and A. Aysan, "Coevolution of Mobile Malware and Anti-Malware", IEEE Transactions on Information Forensics and Security, vol. 13, no. 10, pp. 2563-2574, 2018.

[53] V. Rastogi, Y. Chen and X. Jiang, "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.

[54] V. Rastogi, Y. Chen and X. Jiang, "DroidChameleon", Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13, 2013.

[55] M. Zheng, P. Lee and J. Lui, "ADAM: Automatic and Extensible Platform to Stress Test Android Anti-virus Systems", Detection of Intrusions, Malware and Vulnerability Assessment, pp. 82-101, 2013.

[56] S. Badhani and S. Muttoo, "Evading android anti-malware by hiding malicious application inside images", International Journal of System Assurance Engineering and Management, vol. 9, no. 2, pp. 482-493, 2017.

[57] M. Chua and V. Balachandran, "Effectiveness of Android Obfuscation on Evading Anti-malware", Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy - CODASPY '18, 2018.

[58] Virustotal.com. (2019). VirusTotal. [online] Available at: https://www.virustotal.com/ [Accessed 11 Aug. 2019].

[59] L. Liu, Y. Gu, Q. Li and P. Su, "RealDroid: Large-Scale Evasive Malware Detection on "Real Devices"", 2017 26th International Conference on Computer Communication and Networks (ICCCN).

[60] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu and D. Wu, "Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android", Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15, 2015.

[61] Droidbox, “An android application sandbox for dynamic analysis,” https://code.google.com/p/droidbox/, 2011.

[62] L. K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis,” in Proceedings of the 21st USENIX Conference on Security Symposium, Berkeley, CA, USA, 2012, pp. 29–29.

[63] W. Enck, P. Gilbert, and a. et, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” in Proceedings of the 9th USENIX Conference on OSDI, Berkeley, CA, USA, 2010, pp. 393–407

[64] Andrubis, “A tool for analyzing unknown android applications,” http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzingunknown-androidapplications-2/, June 2012.

[65] Sanddroid, “an apk analysis sandbox,” http://sanddroid.xjtu.edu.cn/.

[66] Owasp.org. (2019). [online] Available at: https://www.owasp.org/images/7/7c/TraceDroid.pdf [Accessed 11 Mar. 2019].

[67] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu and D. Wu, "Towards Discovering and Understanding Unexpected Hazards in Tailoring Antivirus Software for Android", Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15, 2015.

[68] Mathworld.wolfram.com. (2019). Fast Fourier Transform -- from Wolfram MathWorld. [online] Available at: http://mathworld.wolfram.com/FastFourierTransform.html [Accessed 11 Mar. 2019].

[69] Diao, W., Liu, X., Li, Z. and Zhang, K. (2019). Evading Android Runtime Analysis Through Detecting Programmed Interactions.

[70] A. A. Axelle Aprville, "Hide Android Applications in Images," 2014 in Paper presented at BlackHat Europe, Amsterdam, NH.

[71] "https://github.com/," [Online]. Available: https://github.com/ashishb/android-malware. [Accessed September 2019].

[72] "https://www.f-secure.com/," [Online]. Available: https://www.f-secure.com/v-descs/backdoor_android_dendroid_a.shtml. [Accessed September 2019].

[73] "https://blog.virustotal.com/," [Online]. Available: https://blog.virustotal.com/2018/04/meet-virustotal-droidy-our-new-android.html. [Accessed September 2019].

[74] "https://blog.virustotal.com/," [Online]. Available: https://blog.virustotal.com/2017/11/malware-analysis-sandbox-aggregation.html.