https://doi.org/10.1016/j.cose.2015.02.007

Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware

Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, Giorgio Giacinto[a]

[a] *University of Cagliari,*
Piazza d'Armi 09123, Cagliari, Italy

Abstract

In order to effectively evade anti-malware solutions, Android malware authors are progressively resorting to automatic obfuscation strategies. Recent works have shown, on small-scale experiments, the possibility of evading anti-malware engines by applying simple obfuscation transformations on previously detected malware samples. In this paper, we provide a large-scale experiment in which the detection performances of a high number of anti-malware solutions are tested against two different sets of malware samples that have been obfuscated according to different strategies. Moreover, we show that anti-malware engines search for possible malicious content inside assets and entry-point classes. We also provide a temporal analysis of the detection performances of anti-malware engines to verify if their resilience has improved during 2013. Finally, we show how, by manipulating the area of the Android executable that contains the strings used by the application, it is possible to deceive anti-malware engines so that they will identify legitimate samples as malware. On one hand, the attained results show that anti-malware systems have improved their resilience against trivial obfuscation techniques. On the other hand, more complex changes to the application executable have proved to be still effective against detection. Thus, we claim that a deeper static (or dynamic) analysis of the application is needed to improve the robustness of such systems.

Email address: `{davide.maiorca, davide.ariu, igino.corona, giacinto}@diee.unica.it`, `arsu.ma@gmail.com` (Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, Giorgio Giacinto)

Keywords: Android, Malware, Obfuscation, Evasion, DexGuard, Assets, Dalvik, Entry Points, Signatures, Strings, Bytecode

1. Introduction

Not surprisingly, malware writers are paying more and more attention to mobile devices. In fact, the number of mobile devices sold worldwide has already surpassed that of traditional personal computers. According to a recent report by `F-Secure`, more than 99% of the new mobile malware families discovered in 2014 targets the `Android` platform, which accounts for more than 750 millions of active devices [1].

There are several reasons for which Android is a particular interesting target for deploying malware:

1. Its open source nature allows an attacker to carefully studying the operating system implementation, thus increasing the probability of finding vulnerabilities.
2. There are multiple alternative markets besides the official one (`Google Play`), in which it is possible to find applications that are not released through the support of Google (for example, for copyright reasons) or to find popular premium applications at a reduced price. Popular examples are the `Amazon` or `Samsung` app stores ([2, 3]). However, many of these markets provide insufficient control on the security of the applications, thus becoming the first source of mobile malware [4].
3. Even if `Google Play` features an advanced dynamic analysis system for detecting malware and spyware, called `Google Bouncer`, several malware have managed to bypass it even very recently (see, for instance, [5, 6, 7]).
4. The problem gets even more serious because of the bad management of digital signatures in the `Google Play` store. In order to upload an app to the store and install it on the phone, an application must be signed with a certificate. This certificate is also also used as a reference for other applications to *share* their data. Sadly, many legitimate applications are

signed with extremely insecure private keys (i.e., private keys that can be easily obtained by an attacker). It is therefore not so difficult for an attacker to digitally sign a malicious application so that its certificate would look trusted. In this way, it is even possible to *replace* a legitimate app with its malicious variant through an update process [8]. Recently, a dangerous vulnerability that exploited the lack of control in the certificate issuer by the Android cryptographic system has been discovered, and it has been there until the most recent versions of Android, including `KitKat` (4.4) [9].

To improve the security of the users, a number of different anti-malware solutions has been developed. These solutions perform an in-depth scan of the application, including its bytecode, external resources such as images, and audio and so forth etc. Consequently, `Android` malware are getting more sophisticated, not only because of the different types of attacks that they can implement (e.g., root exploits, embedded executables, invisible layouts, encrypted C&C communications, etc. [10, 11]), but also for the possibility of deceiving reverse engineering attempts or anti-malware analysis through *obfuscation* [12, 13]. In this paper, we refer to the term obfuscation as actions that perform changes on the application while preserving its semantics. For instance, they can modify its bytecode, strings, or resource files. The aim of obfuscation is making applications more difficult to be analyzed by humans or automatic tools.

Obfuscation can be used to protect applications from being plagiarized or cloned. However, some obfuscation strategies might also be used to easily create new versions of the same malware that are more difficult to analyze. The attacker is motivated to adopt them, as automatic analysis tools often rely on *static signatures* that can be easily evaded by changing few elements of the applications (for example, replacing the name of the methods). It is possible to find different examples of obfuscation in the wild, such as those reported in [14, 15, 16]. In order to ease malware obfuscation, there is a number of automatic tools, some of them available as a commercial product, and others available for free [17, 18].

Previous works explored the world of obfuscation for the Android platform, by pointing out how specific obfuscation techniques can be effective to evade popular anti-malware solutions [19, 20]. In particular, the work by Rastogi et al. clearly showed how anti-malware systems are weak against easy-to-implement transformation techniques. In particular, it showed that it is possible to evade the vast majority of the most popular anti-malware software for `Android` by applying a combination of obfuscation techniques. It also showed that many anti-malware software tend to rely on signatures that are weak and easy to bypass. That work was carried out by testing obfuscation techniques against 6 malware samples, where anti-malware signatures were updated at the beginning of 2013 [21].

In this paper, we provide a deeper insight into the effects of the obfuscation of `Android` malware. First, our interest is to assess the current status of the anti-malware detection capabilities, as almost one year has passed since the analysis made by Rastogi et al. We do so by deploying a *large-scale* experiment on more than *50* malware families and two malware datasets, namely, `Malgenome` and `Contagio`, for a total of more than `1200` samples [22, 23]. We obfuscate the malware samples in these datasets by means of different strategies, which differs from each other in terms of complexity (from simple ones such as class and methods renaming, to complex ones such as reflection and class encryption), and areas of the `Android` executable that are targeted (e.g., *strings*, *bytecode*, or both). We experiment new obfuscation strategies, and their combinations that have never been tested in previous works (such as the combination of obfuscation by Reflection and Class Encryption). Our tests have been carried out by running `13` among the most popular anti-malware solutions available on the `Android` market `Google Play`. This experimental set-up provides, to the best of our knowledge, the biggest assessment of anti-malware performances against obfuscated samples in comparison with previous works.

The second contribution of this work concerns the assessment of the parts of the application that might be decisive in the anti-malware detection process.

To this end, we focused on the incidence of external resources, such as *assets*. We found that anti-malware engines resort to analyzing and flagging external resources as malicious as an aid for the detection, thus confirming the findings in previous works. However, we also point out the extent to which the analysis of external resources plays a key role in the final outcome produced by anti-malware tools. To this end, we show that by manipulating the assets of a malware sample it is possible to evade the detection. We will also explain, though, that while the ad-hoc manual implementation is quite easy to deploy, its automatic version is quite difficult to develop.

We also examine the role of *entry-point* classes in the detection outcome of anti-malware systems detection. In particular, we show that many anti-malware engines rely on the analysis of such classes in order to perform malware detection. While obfuscating such classes is not trivial, as it might lead to completely break the application, we believe that such a choice is reasonable, to the extent to which other obfuscation strategies fail at evading detection.

Our analysis allows us also to show an interesting trend in the performances of anti-malware products. On one hand, we show that trivial obfuscation techniques, as well as simple combinations of these techniques, are not enough anymore to evade the majority of anti-malware solutions. In one year, signatures and heuristics adopted by such systems have improved, forcing the attacker to perform changes that might end up in breaking the application. Some anti-malware solutions exhibit improved static analysis of the code, as well as a more in-depth analysis of the application entry points. On the other hand, many anti-malware solutions still fail at detecting advanced combinations of these techniques

To further confirm this trend, the third contribution of this paper provides a temporal analysis of the detection performances for some specific samples that have been considered in the past in previous works on obfuscation. We show that, for such samples, the obfuscation techniques needed to evade specific anti-malware solutions require more changes to the executable code and to the resource files compared to the past. From these analysis we can desume that, if anti-malware solutions have shown great improvements compared to the past, they still need further development as they are still vulnerable when they are targeted by complex obfuscation mechanisms.

Additionally, we also point out which obfuscation techniques effectively perform on each anti-malware solution. In particular, we show which strategy allows for obtaining good evasion performances while keeping its complexity as low as possible.

The final contribution of the paper aims at providing an assessment of the easiness of deceiving anti-malware systems. Is it possible to create benign samples that will be considered malicious by anti-malware engines? We show that this result can be easily attained by injecting *malicious strings*, i.e., strings found in malicious samples (but that will be never used in, or called by the code), inside a perfectly benign sample, which will only perform benign operations even after the injection. We show that the majority of the engines exhibit a huge amount of false positives, thus pointing out that those signatures are weak against those attacks. Interestingly, we show that these benign samples are wrongly detected as being of the same type of the malware from which the malicious strings have been extracted, regardless of the actual bytecode, thus showing that strings have a key role in the detection process. This is a serious issue, as the user might not trust its anti-malware product, if too many false alerts are raised, and thus decide not to use anti-malware products. Furthermore, there are practical cases in which Intrusion Detection Systems rely on signatures that are similar to the ones of the anti-malware engines (such as Snort [24]), and this kind of attack might easily confuse them.

The remainder of this document is organized as follows. Section 2 briefly sketches the fundamentals of Android applications and provides a description of the `.dex` file structure. Section 3 describes the obfuscation strategies adopted in this paper, pointing out how they modify the `classes.dex` and the *resource* files. Section 4 presents the experimental evaluation. Section 5 provides a description of the state-of-the-art works related to `Android` obfuscation. Section 6 discusses the experimental evaluation, as well as possible countermeasures in order to improve anti-malware performances. Section 7 closes the paper with our conclusions.

2. The Android Platform

2.1. Android Applications

An Android application is basically a compressed archive with `.apk` file extension. This archive contains:

- **AndroidManifest.xml**, a file with the description of the main application components, i.e., the classes from which the application starts its execution (entry-points), the permissions used by the application (e.g., requesting Wi-Fi of SMS functionalities), the actions used to activate a specific component (intents), etc.
- **classes.dex**, a `Dalvik Virtual Machine` executable obtained by a) compiling the `.java` files that contain all the classes used by the application (thus, generating `.class Java Virtual Machine` files) and b) converting the `Java Virtual Machine` files to `Dalvik` byte code. `Dalvik` is a virtual machine similar to its `Java` counterpart, but optimized for mobile phones, and in particular for systems with limited memory or computational power. It is worth noting that the `classes.dex` file can be disassembled into `.smali` files, a simplified format that facilitates the reading of disassembled bytecode (see, for instance, the usage of `baksmali` [25] to disassemble `Android` executables).
- **Assets**, external resources needed for the execution of the application (e.g., audio files for multimedia applications, images, or, more recently, even executables containing exploits).
- **Resources**, a number of `.xml` files, which describe how layouts (i.e., the visual structure of the user interface), menus, dialogs, etc., are designed.

When an application is started, the AndroidManifest.xml is accessed to extract the *entry-point classes* of the application, i.e., those classes that are *explicitly* declared in the AndroidManifest.xml file, and to read the permissions needed for the execution. These informations are used by the application, for instance, to identify the main class of the application inside the .dex file. Therefore, the entry point classes defined in the AndroidManifest.xml should be coherent with the definitions contained inside the .dex file, in order not to compromise the functionalities of the application.

2.2. DEX Format

2.2.1. Overview

For the purposes of this paper, it is of interest to provide an insight into the modus operandi of the Dalvik (.dex) format. The Dalvik Virtual Machine is a register machine, i.e, the operands on which the instructions operate are stored in registers. This allows a higher optimization when compared to the Java Virtual Machine (which is a stack machine). For example, the instruction c=a+b would be represented in Dalvik as add-int v0,v1,v2;. In a .dex file, there is a unique Data section (at the *end* of the file) which contains information such as classes, fields, names access flags, fields and methods names and, ultimately, methods bytecode. The various parts of the Data section of the .dex file are referenced by IDs, i.e., data structures that contain references to specific parts of the Data section. In this way, it is easily possible to trace methods, strings, and fields to reconstruct elements such as string names or methods return types [26, 27].

2.2.2. DEX File Structure

We will now provide a more detailed description of the elements of the .dex file format [26, 27].

- Header. It is a data structure which contains, in order: **i)** a magic number (i.e., a byte sequence that directly identifies that a .dex file is used), **ii)** a file checksum number, **iii)** a 20 bytes SHA-1 hash of the whole .dex file with the exception of the magic number, the checksum, and the hash itself, **iv)** header and file size, **v)** an endian tag (data in `.dex` files are stored in `little endian` format, i.e., bytes are in reversed order) **vi)** specific addresses of the `Data` area, **vii)** size and position of the different `IDs` areas.

- `String IDs`. They are ordered addresses that point to the `Data` section in which the related strings are stored. It is worth noting that the `ID number` is defined by the position of the address in the section. For example, the first address of the section will be related to the string with `ID Number 0`, the second to `ID Number 1` etc.

- `Type IDs`. They are addresses related to `String IDs` which contain the reference to the corresponding type `String` (for example, a string `L` represents a class).

- `Proto IDs`. They are data structures which contain addresses of strings that, when combined, creates a method prototype. Therefore, they mostly indicate how to find the method return types and parameters.

- `Field IDs`. They are data structures which contain the references to retrieve information about classes fields. In particular, they contain: **i)** their class id, **ii)** their type id (reference to type IDs) and **iii)** their name id (they refer to string ids).

- `Method IDs`. They are data structures which contain the references to retrieve information about methods. In particular, they contain: **i)** their class index, **ii)** their prototype id (they refer to Proto IDs) and **iii)** their name (they refer to string IDs).

- `Classes Defs`. They are data structures which define all the class parameters such as the superclass, the access flags, the interfaces offsets, its bytecode method addresses (i.e., where the method bytecode starts), etc.

- Data. This is the main portion of the .dex file and it is divided into two parts. The first, called class data item, is a data structure that contains all the information related to the size and offsets of static and instance fields, as well as those of virtual and direct methods and annotations. The second part is called code item, and contains the bytecode for each method of each class. For the sake of simplicity, we will refer to the code item section with the term bytecode.

There is also a debug section, that contains some useful information about the source files line numbers, variable names, etc. This section does not contain crucial information on the execution of the application and might as well be removed. From the above description, it is easy to observe that the .dex format is extremely compact, as the IDs mechanism allows for an efficient management of the references. In addition, this format also allows, with some experience, to easily understand the meaning of the different components of the application, and even to decompile it.

3. Obfuscation Strategies

With the term *obfuscation*, we refer to any modification of the Android executable bytecode (i.e., the content of the .dex file) and/or .xml of the files (such as AndroidManifest.xml or resources-related files like String.xml), that does not affect the original functionalities of the application.

The techniques the we adopted can be divided into two sets. One set, that we called Trivial Obfuscation Techniques in agreement with the current literature, contains obfuscation techniques that modify only strings in the classes.dex file. In the other set, that we called Non-Trivial Obfuscation Techniques, we employ techniques that modify both the strings and the bytecode of the executable. For each set of strategies that we adopted, we also include obfuscation techniques that target .xml files, such as AndroidManifest.xml. In fact, there are cases in which obfuscated malware can be still detected because of signatures based on information in the Manifest or other .xml files.

These choices are related to how anti-malware systems perform their detection. The first and easiest way to find anomalies in an application is by *matching* specific elements that are known to be related to malicious activities. For instance, in the case of Android applications, their package and class names (contained in the *string* section) might alone be enough for the anti-malware to decide about their maliciousness. Likewise, strings shown on the screen or used as variables for performing specific operations can be useful indicators for the detection. Anti-malware engines can therefore associate to a malware specific *signatures* extracted from the presence of certain strings inside its executable [19, 21].

Obviously, such signatures might be quite easy to evade, as an attacker would be able to conceal most of the strings with a minimum effort. For this reason, some anti-malware engines also implement *heuristics* based, for example, on the *static* analysis of the code. In particular, they can analyze sequences of *bytecode instructions*, thus trying to identify the presence of malicious operations. This analysis is of course more computationally expensive, but more robust against trivial evasion attempts. Other approaches might combine information retrieved from strings and bytecode instructions.

Another element that might provide contributes to the detection is the analysis of the `AndroidManifest.xml` and of the *resource* files, e.g., `.xml` files that describe the application layout. Concealing information in these files is more difficult, as careless modifications might completely break the application functionalities. Thus, some engines perform a simple `SHA1` check of these files against a blacklist of known malicious ones.

As most of anti-malware engines are not open-source, we do not exactly know which detection strategies they employ to perform their detection. Therefore, addressing different elements of the application is a useful strategy to stimulate all possible detection mechanisms that can be adopted.

It is worth noting that among the obfuscation techniques that we have employed, some of them have been already tested in previous works [21], while other more sophisticated techniques, such as Class Encryption, have only been proposed in previous works from a conceptual viewpoint. In this paper we show how these more sophisticated obfuscation techniques can be actually implemented, and the related experimental results. In addition, we also want to point out that the aim of this work is not to test the largest number of obfuscation techniques, but to investigate a set of *diverse* obfuscation techniques that exhibit different levels of implementation complexity, and that modify different elements of the application. This choice also allowed us to propose novel combinations of obfuscation techniques that were never proposed nor tested before (e.g., the combination of Reflection and Class Encryption). These combinations produced the largest amount of modifications of the bytecode never seen before.

3.1. Trivial Obfuscation Strategies

With the term `Trivial` we define an ensemble of obfuscation strategies that only affects *strings*, without changing the bytecode instructions. This strategy consists in replacing the names of all packages, methods, classes, fields and source files of an `Android` application with random letters. Obviously, such operations include disassembling, reassembling and repacking the `classes.dex` file. These techniques have also been employed by Rastogi et al. on a small number of cases [21]. We note that Rastogi et al. defined as *Trivial* only simple repackaging and disassembling/reassembling solutions. In our definition of *Trivial*, along with such strategies (that we name *Naive* for better clarity), we include what Rastogi et al. called *Transformations Attacks Detectable by Static Analysis*. When such changes are applied to entry-point classes (i.e., classes that extends fundamental functionalities of `Android`, such as activities, broadcast receivers, etc.), the `Android Manifest.xml` file must be changed accordingly, otherwise the application will be broken. We expect these techniques to be effective against anti-malware engine, as many anti-malware engines classify a sample as malware by simply detecting the presence of the names of suspicious classes, packages or methods. This operation leads to changes at strings, fields, methods, classes definitions levels of the executable file structure. In particular, since the same letters can be used to reference both name methods and fields, there may be a reduction of the number of strings within the data section and of their relative references (i.e., the size of IDs changes).

3.2. Non-Trivial Obfuscation Strategies

In this section, we introduce techniques that affect both the strings and the bytecode of the executable. All these techniques were mentioned by Rastogi et al. [21]. However some of these, like Reflection and Class Encryption, have not been extensively analyzed. Such techniques are effective against anti-malware systems that analyze the bytecode instructions to detect malware. Likewise, different types of strings (e.g., constants) are modified, and this might tackle engines which resort to analyze them in order to perform detection.

3.2.1. Reflection

Reflection is the property of a class of inspecting itself, thus getting information on its methods, fields, etc. In particular, `Java` supports such property, by leveraging on the `Java.reflect API` [28]. In this paper, we use the reflection property for *invocations*, i.e., we replace each `invoke` type instruction with a number of bytecode instructions that leverage on reflective calls to perform the same action as the replaced instruction. In this technique, three invocations are used to replace the original one: **a)** `forName`, which searches for a class with a specific name **b)** `getMethod`, which returns the target method object (related to the class name obtained before), and **c)** `invoke`, which performs the actual invocation on the method object that is the result of the second invoke. Typically, the use of reflection would bring to a waste of bytecode instructions. This is the reason why such technique is only used in code development under particular circumstances.

3.2.2. String Encryption

This technique obfuscates every string that is defined inside a class by means of an algorithm based on `XOR` operations. At runtime, the correct string is generated by passing the encrypted string (represented as a byte array) to a function that performs the decryption mathematical operations (it takes, as arguments, three integers). Although this mechanism does not resort to `DES` or `AES` algorithms, it is worth noting that is a more complex mechanism than the ones proposed in the literature for string encryption, which adopted a `Caesar` shift [20].

3.2.3. Class Encryption

This is the most powerful and advanced obfuscation technique adopted in this paper. This obfuscation technique completely *encrypts and compresses* (by means of the `GZIP` algorithm) each class, and stores its data in a *data array*. Consequently, a new method that will perform decryption, and load this class at runtime, needs to be created. Accordingly, during the execution of the obfuscated application, the obfuscated class needs to be first decrypted, decompressed, and then loaded in memory. After that, the methods `getClassLoader()`, `getDeclaredConstructor` and `newInstance()` will create a new instance of the class [29]. Finally, every time the methods or the fields of the class need to be accessed, the Reflection `API` will be used accordingly. This technique can highly increase the overhead of the application as a lot of instructions are added. However, it makes enormously difficult for a human operator to perform static analysis.

3.3. Other Obfuscation Strategies

3.3.1. `.xml` files and resources

We complement the obfuscation of the `classes.dex` file, by performing some additional operations on the `AndroidManifest.xml` file and on other resources-related `.xml` files. This is done for two reasons: **i)** To adapt the `AndroidManifest.xml` to some changes made on the executable file, and **ii)** to undermine the effectiveness of signatures of anti-malware engines that might rely on the `MD5` value related to some resource files that has been found in malware samples. Such changes include, for example, the removal of the `android:name` tag from the `Android Manifest.xml`, as well as the modification of some entry point definitions. Nonetheless, we adapt the removal of specific strings in the `String.xml` files.

3.3.2. Assets

We also perform obfuscation of assets, by means of a simple `XOR` encryption. This is crucial, as many anti-malware engines, in order to detect a malicious application, perform a check on assets. Although easy to break, the employed encryption technique is enough to make the asset non detectable by an anti-malware engine.

3.4. Combining Obfuscation Techniques

The above techniques can be combined in order to make malware detection more hardly detectable. Combinations of different obfuscation techniques have been previously proposed in the literature. However, as many anti-malware systems at that time could be evaded by obfuscating just one of the components of an Android application, few combinations have been tested in previous works. The combinations tested and reported in this paper aim at providing a deeper level of obfuscation compared to single-ended solutions, as they allow for bigger bytecode changes in comparison with previous works (i.e., more instructions are changed).

We used different combinations of the obfuscation techniques described in the previous sections. Some of these combinations (in particular, combinations between `Trivial` and `non-Trivial` techniques) have been already tested in the literature to break detection when using single obfuscation techniques was not effective. It is worth noting that `Trivial` Obfuscation Techniques will always be adopted before `non-Trivial` ones. This procedure avoids possible crashes of the obfuscated Android application when methods or classes are renamed after applying, for instance, Reflection. For the same reason, String Encryption will always be applied before Reflection, and the latter will always be used before Class Encryption. In fact, once all classes are encrypted, no further modifications are possible.

4. Obfuscation Assessment

4.1. Objectives

In this Section we experimentally assess the effectiveness of the obfuscation strategies described in Section 3, as well as the easiness of deceiving the anti-malware detection capabilities. First, we are interested in pointing out which obfuscation techniques allow for evading the largest fraction of anti-malware engines, by also combining the obfuscations approaches described in Section 3. This is done by also providing some insights into the *overhead* that each technique will bring to the application in terms of *size*.

Second, we point out the role of external *assets* with respect to the main application files. In particular, we show how anti-malware engines rely on the analysis of such external files to detect malicious applications.

Third, we analyze the role of the application *entry-points*, by showing how much anti-malware detection capabilities rely on their analysis.

Fourth, we make a comparison between our results and the ones reported in [21]. We performed the tests on the *same* set of malware samples used in [21], to see if the obfuscation techniques with which it was possible to evade anti-malware solutions at the time of the experiments reported in that paper, were still effective. Thus this experiment aims at assessing whether or not the anti-malware detection capabilities have evolved through time.

Fifth, we show, for each anti-malware engine and under the scenarios considered in the previous points (i.e., simple apk obfuscation, encrypted assets and obfuscated entry-points), the *least* complex obfuscation that yields to a detection rate drop of more than 50%. This is done to prove that each anti-malware engine is particularly sensitive to a specific obfuscation strategy and that the optimal choice of the attack depends on the targeted system as well.

Finally, we test the easiness of deceiving the anti-malware detection capabilities. For example, is it easy to trick an anti-malware engine so that a benign sample is considered malicious? We will provide the answer in the next sections.

4.2. Dataset and anti-malware Engines

4.2.1. Datasets

In order to perform the assessment, we used a dataset made up of samples collected from two representative sources of Android malware. The first one is MalGenome [22], a very popular dataset collected by Zhou and Jiang in 2012 [30]. This dataset contains more than 1200 malware samples that emerged in the wild from August 2010 to October 2011. The second one is Contagio MiniDump, a dataset composed by 237 samples collected from the popular malware analysis website Contagio [23]. These malware samples have been collected between December 2011 and March 2013. We have not included recently discovered malware, as detection engines might not have fully updated their signatures to such new releases, and so they can be vulnerable to obfuscation. It is important to note that, to the purposes of our paper, we applied obfuscation strategies to well known samples (i.e., samples for which we expect signatures have been fully deployed) to see if they can still harm Android users by evading anti-malware software.

4.2.2. Anti-malware engines

To perform a deep and significant analysis, we have collected 13 signature-based anti-malware systems. These systems represent the most popular, and the most downloaded ones from Google Play. However, differently from previous works, we are not interested in assessing the performances of a specific anti-malware system, or establishing a particular ranking among these systems, which is something that is already available from other online services (e.g., [31]). For this reason, we will report the attained results in terms of detection statistics over the set of considered anti-malware engines. We believe that this is a more interesting analysis, as a common user might randomly choose between one of the systems that have been tested in this paper. Table 1 shows the anti-malware engines that have been adopted for this analysis, along with their version. All signatures have been updated in February 2014. To the best of our knowledge, this is the largest amount of anti-malware systems ever used

Table 1: List of Antivirus apps included in the experimental evaluation.

Vendor	**Version**	**Vendor**	**Version**
Avast	3.0.7118	AVG	3.5.1
Comodo	2.4.1	Dr.Web	9.00.1
ESET	2.0.853.0-15	Fsecure	8.3.14209
GData	24.5.4	Kaspersky	11.2.4.105
McAfee	3.2.0.2193	Norton	3.8.0.1199
TrendMicro	3.5.0.1348	WebRoot	3.5.0.6058
Zoner	1.8.2		

in a mobile assessment of this kind.

We also point out that we did not resort to services such as `VirusTotal` or `AndroTotal` [32, 33]. Despite them being useful services, they have their own limitations for the purposes of this paper. By using `VirusTotal`, we would have leveraged on X86 anti-malware engines that are not specifically developed for mobile applications, and therefore might not be accurate as their mobile counterparts. In addition, we included in our testing environment a number of engines which is twice the number of those featured in `AndroTotal`, and, more importantly, we had complete control on the engine versions, which is crucial for a fair evaluation. For this reason, every engine has been installed and run on *physical* devices featuring `Android 4.2.2`.

4.3. Experimental Protocol

In the following, we report the results of four sets of experiments, each aimed at assessing the effectiveness of obfuscation from different viewpoints:

- **General obfuscation assessment**. In this experiment we obfuscate the whole `Contagio` and `MalGenome` datasets by means of the techniques described in Section 3. We tested *seven* different obfuscation scenarios. In the first four scenarios, each of the techniques described in Section 3, namely, Trivial techniques, Reflection, String Encryption, and Class Encryption, is used stand-alone. In the last three scenarios, the following different combinations of these techniques are used: **i)** Trivial techniques followed by String Encryption; **ii)** Trivial techniques followed by String Encryption and Reflection; **iii)** Trivial techniques followed by String Encryption, Reflection, and Class Encryption.

Malware obfuscation has been carried out by resorting to the commercial tool `DexGuard 5.5` [18] which is, to the best of our knowledge, the most powerful obfuscation tool for `Android` applications that is publicly available.

- **Extended obfuscation assessment**. In this experiment, we tested the same scenarios as in the first set of experiments with the addition of the obfuscation of either **a)** assets or **b)** entry-points. We performed experiments separately for the two additional components to be obfuscated, thus resulting in a total of 14 new scenarios. In both cases, `Dexguard` does not provide reliable routines that allow the applications being fully functional after being obfuscated. In fact, after obfuscating assets and entry points, new functions need to be included in the application that allow the obfuscated assets and entry points to be used at runtime. We thus wrote by ourselves the routines that allowed the obfuscated applications to be fully functional.

- **Temporal comparison**. This experiment is aimed at comparing the performances of anti-malware software using the same samples adopted in the experiments reported in [21]. In particular, we reproduced the same obfuscation techniques reported in that paper, and see if anti-malware software are still vulnerabile. In other words, we assessed how anti-malware detection capabilities have evolved through time.

- **Single Anti-malware Evaluation**. In this experiment we show, for each anti-malware engine and under the scenarios of the previous experiments, the least complex obfuscation that yields to a detection rate drop of at least 50%. This is done for two reasons: **a)** To understand if different anti-malware engines are particularly sensitive to specific obfuscation strategies and **b)** to verify if introducing assets and entry-points obfuscation can reduce the complexity of the obfuscation technique that is needed for the evasion.

- **Anti-malware deception**. In this experiment, we assessed the dependance of the detection capabilities of anti-malware engines on the String section of the `classes.dex` file. To this end, **i)** we considered one benign application, **ii)** we generate new *benign* samples by simply injecting, inside the application, strings contained in malicious samples. Such strings are never going to be used, as the benign bytecode is not changed, so the application is *benign* even if the analysis of the String section may drive to the conclusion that the application is malicious. The aim of this experiment is to verify to what extent malware detection is triggered by the strings in the applications. If this is true, the *benign* samples crafted according to the above procedure can be used to generate *false positives.*

It is worth noting that for each experiment, to avoid the detection rate being influenced by the content of `.xml` files such as `AndroidManifest.xml`, changes like the ones described in Section 3 are always performed in order not to trigger signatures based on the `SHA1` value computed on `.xml` files.

4.4. General Obfuscation Assessment

In this experiment, we obfuscate the malware dataset according to the Experimental Protocol described above. In this way, we created *seven* obfuscated datasets, and we used them to test all the considered 13 anti-malware solutions. Figure 1 shows, by means of a box plot, the statistics of the detection rate for the set of anti-malware systems for each of the seven obfuscation scenarios considered.

The box plot is structured as follows:

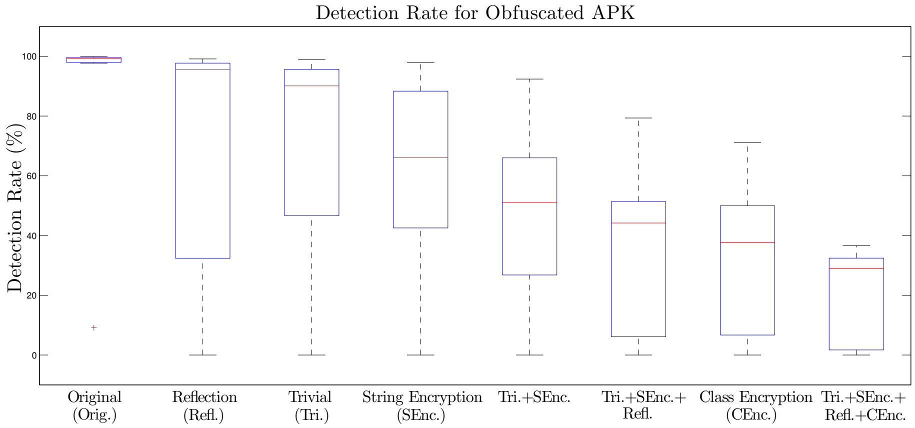

Figure 1: Statistics of the Average Detection Rates of the 13 anti-malware solutions when the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

- On the X axis we represent the obfuscation techniques adopted. On the Y axis, we represent the *average* detection rate of the engines, i.e., the detection rate calculated on the whole dataset D_a. Such value is calculated, for each engine, as:

$$D_a = \frac{N_d}{N}$$

  where N_d is the number of *detected* samples by the engine and N is the number of total samples of the dataset.

- The lower edge of the box represents the first *quartile* H_f of the anti-malware average detection rate distribution, i.e., 25% of the anti-malware engines exhibits an average detection rate below this value.

  The red line represents the *median* M of the distribution, so that 50% of the anti-malware engines exhibits a performance below that value, while 50% is above that value.

  The upper edge of the box represents the *third quartile* H_t, i.e., 25% of the anti-malware engines exhibits an average detection rate above this value.

- The dotted line represents the so-called *whiskers* H_w of the plot, i.e., the distance between the minimum/maximum of the distribution and the first/third quartile (25% of engines are located on the whiskers). Their maximum size $H_{w_{max}}$ is given by:

$$H_{w_{max}} = IQR * 1.5$$

  where IQR is the *interquartile range*, i.e., the height of the box, expressed by:

$$IQR = H_t - H_f$$

- The red dots represent *outliers* H_o, i.e., anti-malware solutions whose average detection rate falls outside the whiskers. Therefore, the condition for obtaining an outlier is:

$$H_o > H_w$$

We now describe, in more detail, the results plotted in Figure 1:

- Almost all of the engines (except for one outlier, `WebRoot`) are able to correctly detect almost all the samples.

- Changing the code by using Reflection is not effective for many anti-malware engines, as the median M of the detection rate is quite high (around 90%). This means that 50% of the anti-malware solutions provides very high performances against this technique. The remaining 50% is distributed in this way: 25% of them shows a detection rate between 90% and 35% (this is obtained by observing the distance between the median M and the first quartile H_t). The remaining 25% shows a detection rate under 35%, thus providing poor performances. The column Reflection therefore shows that, albeit many anti-malware solutions are resilient to this strategy, there are few that are extremely sensitive to it. This means that some engines might only rely to static analysis of the code in order to perform detection.

- By analyzing the Trivial column, we notice that Trivial techniques are not very effective at evading anti-malware engines. More than half of the tested anti-malware solutions exhibit a detection rate that is very close to 100% (i.e., the median M is close to 100%), suggesting that the usage of simple obfuscation techniques is not effective anymore to bypass many anti-malware systems. Another 25% has a detection rate between 50% and 90%, and this is indicated by the space between the median M and the first quartile H_f. We also notice that few anti-malware systems (less than 25%) exhibit a detection rate that is lower than 50%, and this is pointed out by the presence of a lower whisker.

- Using String Encryption reduces the median value M when compared to the the Trivial column. This means that the maximum detection rate value for 50% of the anti-malware engines is now reduced by around 30%, thus pointing out that some engines base some of their detection heuristics on the presence of strings that are not related to package or classes names.

- The highest detection rate drop, i.e, the decrement in height of the third quartile H_t (and, therefore, of the upper side of the box) is observed when Trivial obfuscation techniques and String Encryption are combined. In particular, we observe an average detection rate reduction of about 50% for the 75% of the anti-malware systems. This is interesting, as changes in bytecode are quite minimal but, on the other hand, Strings play different roles in an Android application, so their encryption may have a huge impact. As we explained in the previous sections, Trivial techniques address strings that are used for identifying classes, methods and fields, whilst String Encryption hides all the constant strings, that are then decrypted at run-time. This is another hint that suggests that the detection capability of anti-malware engines strongly depends on the String section.

- Adding Reflection to the previous combinations lowers the first quartile H_f. This means that the maximum detection rate for the 25% is lowered

Table 2: Average percentage increment of the obfuscated applications size

Technique	**Size Increment (%)**
Trivial	-17.17
String **Enc**ryption	12.47
Reflection	44.66
Class **Enc**ryption	194.23
Tri. + SEnc.	-4.68
Tri. + SEnc. + Refl.	55.58
Tri. + SEnc. + Refl. + CEnc.	197

almost to zero. This point is in line to what we observed using Reflection alone, i.e., some engines are particularly sensitive to this strategy.

- Class Encryption is more effective than the combination of Trivial techniques, String Encryption, and Reflection. In Figure 1, we observe how its stand-alone adoption allows for obtaining slightly superior performances with respect to the former obfuscation strategies (a slightly lower median value compared to the column immediately to the left). It is worth noting, though, that Class Encryption is the most invasive and complex obfuscation that we adopted in this paper. So, while it allows evading a large number of anti-malware engines, it also introduces a massive overhead in terms of file *size*. Table 2 shows the average file size increment in the case of obfuscation by Class Encryption, and in the case in which the combination of Trivial, Reflection and String Encryption is used.

  This table shows that there is a *file size decrement* (negative number) when Trivial techniques are used. This is expected, as most of the times original strings are replaced with strings one- or two characters long. On the other hand, it is very interesting to point out the huge file size increment introduced by Class Encryption. On average, such increment is almost 200%, which means that the size of the application can increase up to four times with respect to its original size.

Summing up, the above results allow us to conclude that an attacker might attain a good evasion rate, with minimum size increment, by employing the Trivial+String Encryption strategy. The use of more complex obfuscation techniques, i.e., the combination of three obfuscation techniques or the adoption of Class Encryption, provides small improvements in evasion, while making the application size strongly increase.

The combination of all the four obfuscation techniques considered in this set of experiments reduces the size of the whiskers, which means that all the engines exhibit a detection rate between, in this case, 0% and 35%. This implies that applying all the four strategies is effective against *any* anti-malware engine. The median value M indicates, though, that 50% of the engines still detects around 30% of the samples.

4.5. Extended Obfuscation Assessment

4.5.1. Assets Obfuscation

As observed in the previous experiments, part of the anti-malware engines still exhibits a detection rate of roughly 40%, even when all the obfuscation techniques are combined.

For this reason, is it of interest to understand what still triggers anti-malware engines to raise an alert, thus keeping the detection rate of half of them around 40%. An interesting hint is given by focusing the analysis on the *less evasive* malware families, i.e., those malware families that can still be detected by the majority of the anti-malware engines when all the four obfuscation techniques are applied. Table 3 shows such families, along with their *average* detection rate attained by employing the combination of all the obfuscation strategies, ordered from the least evasive to the most evasive one.

All the applications belonging to these families, with the exception of `JSMS Hider`, have in common the presence of *assets*.

By individually testing each of the files belonging to the *assets* on the anti-malware engines, we realized that they were flagged as malicious. When included in a zipped archive, such as an `.apk`, this would result in flagging the whole apk,

Table 3: List of the most evasive families. The Average (Avg.) Detection Rate (DR) is reported

	Family Name	**Avg. DR (%)**		**Family Name**	**Avg. DR (%)**
1	Zhash	61.54	6	JSMSHider	53.85
2	Asroot	55.77	7	BaseBridge	53.15
3	DroidDeluxe	53.85	8	DroidKungFu2	50.26
4	DroidDream	53.85	9	DroidKungFu1	46.83
5	GingerMaster	53.85	10	AnserverBot	46.4

as malicious, despite the `.dex` and the `.xml` files being obfuscated and, therefore, undetected. The role of such assets in triggering alerts was already pointed out in the literature [21]. However, our experiments were aimed at better evaluating the impact of such assets on the average detection rate. We argue that this is a crucial point for an effective obfuscation strategy aimed at completely evading malware detection.

Although encrypting the asset file, for example using `XOR` operations, is rather straight-forward, *decrypting* them at run-time is not an easy task. `Dex Guard` does not provide a reliable support for including decryption routines of assets. We therefore developed a technique that can be reliably applied for decrypting such assets, and we also developed a number of proof of concepts related to various families, thus allowing us to assess that the proposed obfuscation strategy can produce a working sample. The proposed technique can be summarized as follows: **i)** Each asset is opened by using the method `open` of the `AssetManager` class. This method will return an `InputStream` that will be converted in a byte array, and then it will be written as a file in a location when it can be then executed with the command `exec`. **ii)** We *disassemble* the `classes.dex` executable by means of `Baksmali` [25] and we *intercept* the byte array that is created. Then, we inject the decryption method inside the disassembled class and we use the intercepted byte array as the method parameter. Such method, at runtime, will return a new, decrypted byte array that will be written instead of the encrypted one. Then, we finally reassemble the whole sample.

To see the effectiveness of Asset Obfuscation, we have performed the same experiments shown in Figure 1 but this time, for each experiment, we also obfuscated the Assets. Figure 2 shows the attained results.

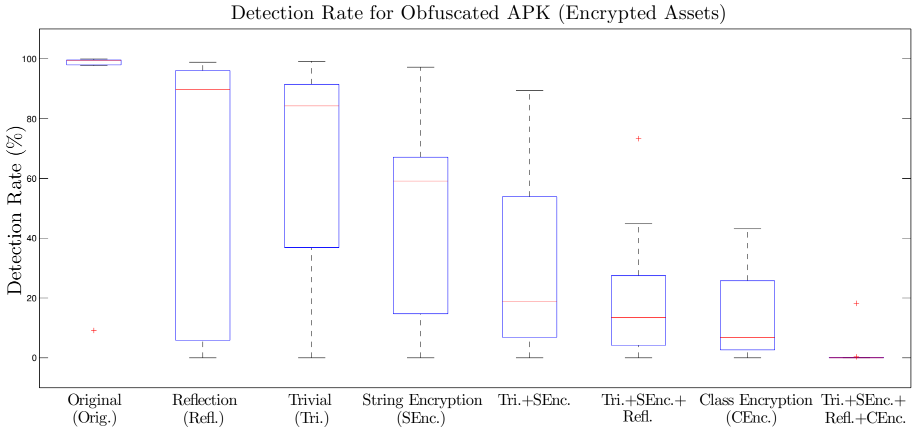

Figure 2: Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

In order to provide a better understanding of this Figure, we will describe it by comparing it to Figure 1. The first thing we notice is that the trend expressed by Figure 2 is basically the same as the one indicated in Figure 1. Thus, all the observations we made about Figure 1 are still valid and the reader can refer to them to understand the trend of this Figure. This means, for example, that the best performances are obtained when combined obfuscations are adopted and when Class Encryption is employed. Likewise, Reflection and Trivial techniques alone are not useful against at least half of the engines. However, we also point out the following differences with respect to Figure 1:

- The median value M gets significantly lower when Trivial and String Encryption techniques are combined. Likewise, the first quartile H_f gets decreased. Therefore, for half of the anti-malware engines, the detection rate gets significantly reduced. This means that the median value in the same column of Figure 1 was higher because the detection rate was influenced by the presence of assets. This is an important point, as combining Trivial and String Encryption techniques is even more effective than what it seemed to be at a first analysis. However, we also observe that the position of the third quartile has only slightly decreased. This means that there is another 25% of anti-malware engines that are resilient to this combination of obfuscations.

- With respect to the previous point, employing Reflection in addition to the previous techniques or Class Encryption alone will also reduce the height of the third quartile H_t, thus reducing the whole box size. This means that, in Figure 2 and for the combination with Reflection, 50% of the engines exhibit a detection rate of 25% and the other 50%, including the whiskers, arrive to 35%. Interestingly, there is one engine (`Comodo`) that does not seem to be influenced by that and it is the outlier with a 70% detection rate.
- Combining all the obfuscation techniques brings the detection rate of all the engines to zero, except for one (`TrendMicro`). This indicates that combining all the obfuscation strategies, while removing at the same time possible external interference, is very effective to bypass almost all anti-malware systems.

4.5.2. Entry points obfuscation

Assets obfuscation significantly reduces the detection rates of anti-malware engines, but some anti-malware systems are still capable of detecting a subset of malware samples, even after applying the most complex obfuscating transformations we have seen so far. There is one other component of an application that we have not tried to obfuscate yet: *entry-point classes*. In this third experiment, we add the encryption of entry-point classes to the experiments reported in the previous section.

Of course, in order for an application to run, the modification of entry-point classes requires some specific changes in the AndroidManifest.xml file. As DexGuard provides limited support for the modifications of the Android Manifest.xml file, we have implemented different proof of concepts. In a similar way to the case of *assets* encryption, we replaced the file names of the classes in the AndroidManifest.xml file with their transformed ones. We were able to prove that it is possible to make a fully working malicious sample even after obfuscating the entry-point classes. Such a mechanism can be easily automated but, in some cases, manual intervention might be required, in particular to handle *malformed* files. It is worth noting that, to further improve the obfuscation process, we make all packages collapse to a single one. This makes also easier to modify the Android Manifest with the correct package.

Figure 3 shows the result for this analysis.

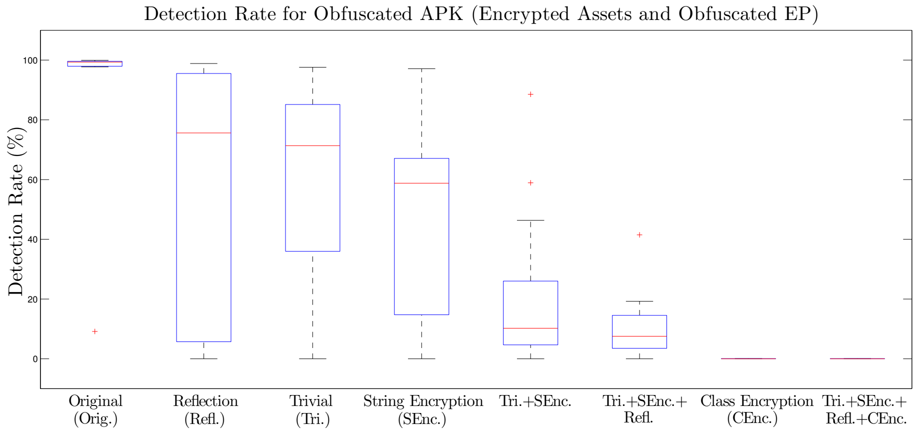

Figure 3: Statistics of the Average Detection Rates of the 13 anti-malware solutions when Assets and Entry Points are encrypted, and the seven obfuscation scenarios are applied. Results are reported in terms of increasing effectiveness of the employed obfuscation techniques

Like we did when we described assets obfuscation, we will compare Figure 3 with 2. Again, obfuscations are increasingly effective in the same way as Figures

1 and 2. Thus, Reflection is still the less effective strategy, while combining obfuscations leads to excellent evasion results. Additionally, we point out the following points:

- Median values get generally lower when adopting Reflection, Trivial and String Encryption techniques (in their stand-alone variant), while the first and third quartile positions remain basically the same. This means that acting on entry-point classes with this strategies influence the maximum detection rate value of half of the engines.
- A huge drop both of the median values (until 10%), as well as of the first and third quartile, is observed when Trivial and String Encryption are combined. This suggests that most of anti-malware engines base their detection on considering a *combination* of different types of strings that often reside in entry-point classes. This is an interesting choice from the viewpoint of anti-malware developers, as trying to obfuscate such classes requires to carefully adapt the `AndroidManifest.xml` file and, therefore, it is an operation that can be hardly automated, as it could lead to damaging the functionality of the application. We also observe two outliers, i.e., `DrWeb` and `Comodo`.
- Applying Reflection in combination with the techniques in the previous point further reduces the whiskers size (this mean that the maximum detection rate value for all the engines is around 15%). An outlier, `Comodo`, still exhibits a detection rate of 35%.
- Applying Class Encryption to entry point classes completely nullifies the detection rate of all anti-malware engines. However, we have also noticed that the obfuscation of entry-point classes and the use of Class Encryption break the functionalities of the application, regardless of the modifications made to the`AndroidManifest.xml` file. In order to have a still working application after applying the Class Encryption transformation, entry-point classes should not be compressed or, alternatively, their decompression

should be done by some external classes/methods.

4.6. Temporal Comparison

After having explored different transformations that allowed evading anti-malware engines, Tables 4 and 5 present a comparison between the obfuscating transformations that were needed to evade anti-malware systems in 2012 and 2013, and ones required in 2014. These tables are based on a similar table reported in [21] where evasion efforts have been compared for the years 2012 and 2013. For each malware sample that has been tested, and for each anti-malware engine considered, we use this notation: `SuccessfulObfuscation2012->SuccessfulObfuscation2013->SuccessfulObfuscation2014`. SuccessfulObfuscation2012 and SuccessfulObfuscation2013 represent the obfuscation transformations that were successful in 2012 and 2013, respectively, according to [21]. For SuccessfulObfuscation2014, that are related to the experiments reported in this paper, we indicate the *less invasive* obfuscation transformation required to make the malware sample not detectableby anti-malware engines. We marked in **bold** SuccessfulObfuscation2014 if *more changes to the executable code or resources are now required to evade the anti-malware system* compared to the past. For better clarity, if the obfuscation strategies that were successful in one year were successful the year before, we used the symbol *. We have adopted the following criteria to choose the anti-malware engines in this analysis.

- Only the anti-malware systems in common between our work and the previous work were adopted. That is a total of 8 anti-malware systems.

- Out of these eight solutions, we chose not to consider `Zoner` and `Webroot`, as these tools can be easily evaded by trivial (or naive) obfuscation transformations. In addition, they also exhibited some problems in detecting the original samples as being malware.

- We also decided not to include `ESET` as the obfuscating transformation used in the previous work in order to evade it in 2012 and 2013 is not part

Table 4: Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (First set of Malware Samples) - See text for explanation of notation

Antivirus	**DroidDream**	**Geinimi**	**FakePlayer**
AVG	Tri.→ *→ **CEnc.+AE**	Tri.→ *→ ***+SEnc.**	Tri.→ *→ ***+AE**
Symantec	Naive→ Tri.→ **CEnc.**	Tri.→ *→ **CEnc**	Tri.→ *→ **SEnc**
ESET	AE→ Tri.+*→ **Refl.**	SEnc.→ *→ **Tri.+*+Refl.**	Tri.→ *→ ***+SEnc**
Kaspersky	AE → *+StrEnc.→ **Tri.+***	Tri. → *→ ***+SEnc.**	Tri. → *→ ***+CEnc.**
Trend M.	AE → *+Tri.→ **CEnc.+AE**	Tri.→ *→ ***+SEnc.**	Nai.→ *→ **Tri+EP**

of Dexguard, and therefore we were not able to reproduce it to verify if changes have effectively occurred during one year.

With such considerations, we restrict our analysis to the 5 anti-malware systems reported in the Tables. We believe that the evolution in the detection capabilities of these systems can clearly show the global trend in anti-malware performances. In order not to generate confusion in the reader, we will use the notation proposed in this paper to specify which obfuscation techniques were used to bypass the systems. It is worth nothing that most of the successful obfuscation techniques in the past fall in the *Trivial* category. If there is an obfuscation technique used in the past, but that was not adopted in this paper, we will explicitly mention it. We also refer to Naive strategies (see Section 3) as **Nai.**, to Encrypted Assets as **AE**, and to operations on Entry Points as **EP**. In one case, files were renamed in the obfuscation performed in the previous work, and we refer to this case as **FR**.

Reported Results clearly show that while in the past it was possible to evade anti-malware engines by resorting to trivial obfuscation strategies, this is not possible anymore. This means that the complexity of the transformations that should be made to the code in order to evade detection is much higher than before, and that malware signatures, along with detection heuristics, have significantly improved during one year. This is in line with the experimental results we have reported in the previous sections. Although it is still possible to evade

Table 5: Evolution of the anti-malware robustness to obfuscation from 2012 to 2014 for specific samples (Second set of Malware Samples) - See text for explanation of notation

Antivirus	**Bgserv**	**BaseBridge**	**Plankton**
AVG	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ **SEnc.+AE**	Tri.→ *→ ***+EP**
Symantec	Tri.+SEnc.→ *→ **CEnc.**	Nai.→ SEnc→ *	Nai.→ *→ **SEnc**
ESET	Tri.→ *→ **CEnc.**	AE.→ *+SEnc.→ *	Nai.→ Tri.+SEnc.→ ***+Refl.**
Kaspersky	Tri+SEnc. → *→ **CEnc.**	Tri.+SEnc. → *→ ***+AE**	Nai. → *→ **Tri.+SEnc.**
Trend M.	Nai. → Tri.→ **CEnc.+AE**	AE→ *+FR→ ***+Tri.**	Nai.→ *→ **Tri+SEnc.**

anti-malware engines, the effort that the attacker has to produce is considerably higher.

4.7. Single Anti-malware Evaluation

Although it is not our aim to provide a ranking of the anti-malware engines, we believe it is useful to observe which obfuscation strategy particularly affects a specific anti-malware system. For this particular test, assuming that the combination of all the obfuscation techniques is obviously the most effective one and that the detection rate of the original samples is almost 100% for all the engines, we will consider the *less expensive, in terms of complexity,* obfuscation that will reduce the detection rate under 50%. This is because, in order to evade a specific anti-malware solution, an attacker could try to find a balance between the effectiveness of the complexity of the adopted strategy. We also want to see if an attacker, by adding further constraints, can reduce the complexity of the chosen obfuscation technique. Table 6 shows the chosen obfuscation for each anti-malware product considered in our evaluation, where the chosen obfuscation is denoted by using the groupings and the same labels as in Figures 1, 2, 3 (e.g., plain `.apk` obfuscation, encrypted assets, encrypted assets and entry-points). We will use the same notation as in Tables 4 and 5. For a better readability, we will use the character * if the technique has not changed from the column immediately to the left. To avoid confusion and for the sake of the clarity, we

Table 6: Less complex obfuscation techniques that will bring detection rate under 50%, under different constraints and for each anti-malware engine

Antivirus	**Apk Obf.**	**Enc. Assets**	**Enc. E.Points**
Avast	**CEnc.**	**Tri.+SEnc.**	*
Comodo	**CEnc.**	*	**Tri.+SEnc.+Refl.**
ESET	**Tri.+SEnc.+Refl.**	*	**Tri.+SEnc.**
GData	**Tri.**	*	*
McAfee	**Tri.+SEnc.**	**SEnc.**	**Tri.**
TrendMicro	**CEnc.**	**Tri.+SEnc.**	*
Zoner	**Tri.**	*	*
AVG	**Tri.+SEnc.+Refl.**	**Tri+SEnc.**	*
Dr. Web	**Refl.**	*	*
Fsecure	**CEnc.**	**Tri.+SEnc.+Refl.**	**Tri.+SEnc.**
Kaspersky	**CEnc.**	**Tri.+SEnc.**	*
Norton	**Tri.+SEnc.**	*	*
WebRoot	**Tri.**	*	*

also do not report the exact percentages drop in the detection rate for each anti-malware systems, and for each obuscation technique. We stress that our aim is simply to make a comparison between the different obfuscations under certain constraints and for each anti-malware system.

From these results it is interesting to see that, in order to affect the performances of some anti-malware systems, advanced techniques such as Class Encryption are normally required. We also point out that, when assets or entry-point classes get obfuscated, the complexity of the obfuscation required is generally reduced. When entry-points are obfuscated, it is possible to decrease the anti-malware engines performances without employing Class Encryption (with the exception of `Kaspersky`).

4.8. Anti-malware Deception

In the previous experiments, we have shown a strong correlation between the strings contained in the `Strings` section of the `classes.dex` file, and the anti-malware detection capabilities. Now, we would like to bring this analysis one step further. The question we want to answer is the following: *how robust are the signatures of the anti-malware engines?* In particular, we want to evaluate how much the anti-malware engines signatures are dependent on the `String` section. This can be done by *injecting* all the strings contained in a malicious sample on a entirely *benign* sample, designed by ourselves. Such strings will never be used or called inside the code. This is done to check if anti-malware engines will consider the sample as malicious even if it does not perform any malicious actions. In other words, we are looking for the possibilities of polluting anti-malware outcomes with *false positives.* This analysis is useful as there are components of a computer security infrastructure, such as IDS, that generate alerts by analyzing application signatures in a very similar way to what anti-malware systems do. Although we are not directly attacking IDS systems in this paper, we believe that this is a good application in which a lot of false alarms can be raised in order to fool an analyzer.

We perform this experiment with the whole `MalGenome` and `Contagio` data sets. In particular, we use *one benign* application as a base and we extract, for each malicious sample in the dataset, *all its strings.* Then, we inject them into the base, thus creating a new sample that will not exhibit any malicious behavior, but will contain Strings belonging to malware samples. Figure 4 shows a comparison between the detection rate of the original base-samples (benign) and the detection rate after string injection, i.e., the fake detection rate.

From this Figure, it is possible to see that half of the anti-malware engines have detected the fake malware with a detection rate up to 80%, whilst the other half spans from 80% to 100%. Interestingly, the assigned malware label is *exactly the same* as the original one. It is worth noting, though, that apart from two engines that resort to the static analysis of the bytecode (and that therefore are not affected by such injection), other three engines exhibit

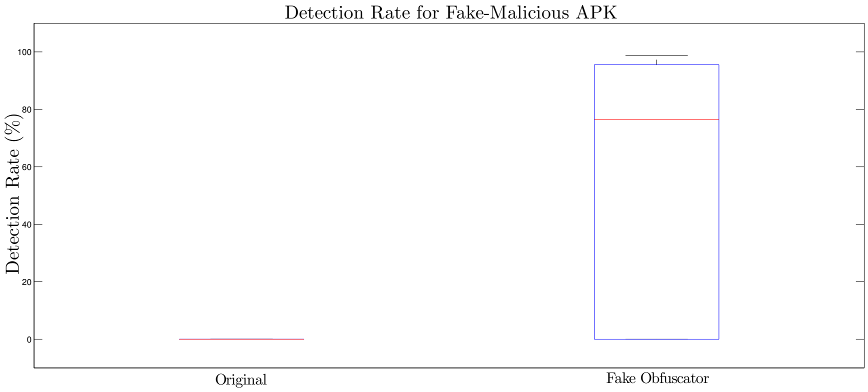

Figure 4: Detection rate of anti-malware engines when strings belonging to malicious samples are injected into benign samples

poor performances even when detecting obfuscated malware, by losing almost all their detection power even when using Trivial obfuscations. Therefore, we can safely conclude that the engines that best perform in detecting obfuscated malware resort to extremely weak detection logics. An attacker could resort to this strategy to make the user lose his confidence on the anti-malware engine itself, due to the high number of false-alarms that might be raised. This result is in agreement with other results in the literature [21]. Apparently, though, compared to the results of the past year, some signatures have evolved, as they include the analysis of the `AndroidManifest.xml` `SHA1` in combination with the analysis of the `.dex` file, as well as an improved analysis of the embedded assets.

5. Related Work

Obfuscation in the Android ecosystem has recently caught the interest of the research community. A comprehensive review of all malware present for the Android ecosystem, as well as their characteristics, has been provided by Zhou and Jiang [30]. Zheng et al. [19] proposed `ADAM`, an obfuscator that performs simple changes on the `Dalvik` executable (e.g., methods renaming, simple changes in the CFG, constant string encryption, etc.). A set of malware was obfuscated with this tool and the capabilities of anti-malware systems in detecting modified samples was tested. Rastogi et al. [20, 21] made similar tests with DroidChamaleon, an extended framework for obfuscating Android applications. Compared to ADAM, such framework provides more obfuscation options (e.g., classes and fields renaming, package names renaming, etc.). This work can be considered to represent the state-of-the-art of anti-malware assessment for Android systems. Another interesting assessment is the one made by Huang et al., in which the resilience of repackaging detectors against obfuscation has been evaluated [34]. Protsenko et al. tested some bytecode obfuscation strategies against anti-malware [35]. All the obfuscation strategies mentioned in this paper are part of the ones proposed by Collberg et al. [36]. In order to provide a better test bench for anti-malware performances, Maggi et al. introduced AndroTotal [33], an online service with which it is possible to scan a malicious application by means of multiple anti-malware systems. Conceptually, the service is the same as VirusTotal [32], but solely focused on Android. Recently, an anti-malware system based on machine learning techniques (Drebin) has been proposed [37].

Some obfuscators were also proposed outside the academic community. Proguard [17] is included in the Android SDK and provides basic obfuscation options. DexGuard [18] is its commercial version, exclusively developed for Android, and it provides advanced functionalities such as *reflection*, *class and string encryption*, etc. Among other Android obfuscators, we mention DashO [38], DexProtector [39], and Allatori [40]. Apkfuscator [41] is a tool that obfuscates Android applications in order to evade specific decompilers, such as Androguard [42], Dedexer [43] and Baksmali [25].

6. Discussion and Countermeasures

From the assessment we have carried out, it is clearly evident that anti-malware engines have significantly improved compared to the past. However, the problem is still clear and present. To be more specific, an attacker would still be able to automatically obfuscate an *entire dataset* and therefore attack his victims with different families, without even being spotted by an anti-malware engine. We have also shown, with an extended assessment, that strings are still used as a mean to build signatures. Although this can be effective if such strings are contained in entry-point classes, this can be a huge drawback when strings from other classes are took into consideration. We therefore discourage the usage of such strategy, unless it is combined with the analysis of some other resources. For example, the analysis of the `AndroidManifest.xml` file could be a better source of information in order to retrieve basic class names and permissions used. Even this analysis, of course, could be evaded, but it strengthens the detection capabilities. Likewise, an analysis of instruction *sequences* can be helpful to detect some malicious behaviors. This also translates into the need of developing some specific *heuristics* that can improve the quality of the application scanning. It could be also useful to analyze *annotations, debug information* or other specific parts of the `classes.dex` file that are sometimes overlooked by an attacker, especially when applications are repackaged.

Class encryption seems to be the best solution for an attacker, but that introduces a big overhead in the application size and execution, and thus might not be the optimal strategy to evade a system. However, if such techniques are used, we suggest that anti-malware engines deploy some dynamic heuristics that, although computationally expensive, might allow for a dynamic dump and decryption of the class, which might most likely show evidence of malicious behavior. An example of this analysis is the one that is performed, for example, by Google `Bouncer` [44] and by other systems like `Anubis` [45]. Such systems execute the Android application in a *virtualized environment* and extract different elements, such as *system calls*, *network traffic*, *services used*, and so forth. In this way, all static evasion attempts are overcome, as only the application *behavior* is analyzed. However, such analysis might require a lot of time to be performed, e.g., *several minutes/sample* to provide significant results, especially for application with a lot of lines of code and services to be called.

It also usually requires more computational resources in comparison to static solutions. For this reason, some systems perform their analysis on dedicated servers and remotely provide the results to the user. However, if such servers are filled with requests, there could be further slowdowns, and even hours might be then required to analyze a single sample. A client-oriented solution, like an anti-malware system, better fits the needs of the user to have a proper answer in a very short time (usually, less than a minute/sample). It is interesting to observe that dynamic analysis is also vulnerable to evasion attempts (see, for instance, [46]). Static anti-malware solutions have also improved their signatures so that, in some cases, they can recognize obfuscation attempts, even without having knowledge of the specific malware. Examples are signatures such as `Crypt3.BBOK` or `Gen:Trojan.Heur.Ey1@ruWJdYoi` that are associated to malware in the wild and they most likely employ obfuscation techniques.

7. Conclusions

In this paper, we have provided a deep insight into the efforts needed to obfuscate `Android` malware, in particular when targeting anti-malware engines. We have tested several obfuscation techniques, which operate on different parts of the `classes.dex` file, and differentiate each other in terms of complexity. We carried out our tests on a large number of samples coming from two popular datasets. We showed that, one year after the last assessment [20, 21], anti-malware engines have significantly improved their performances. Hence, the number of changes that should be made to the executable in order to evade anti-malware systems is now significantly higher.

Our experiments pointed out the decisive role of specific *strings* in the detection capabilities of anti-malware engines. We argued that an attacker can generate an obfuscated sample, with minimum overhead in file size and application performances, by carefully understanding how signatures are designed to identify malware. On the other hand, we also showed that anti-malware engines are protecting from obfuscation attacks by including in their analysis other components of the .apk file, such as *assets*, and *entry-points*. Thus, effective obfuscation mechanisms should keep these components into account to evade detection.

However, although anti-malware performances have improved in 2013, they still exhibit some weaknesses in the detection of complex obfuscation mechanisms, such as the ones made up of combinations of different techniques, or the ones that resort to class encryption. We also showed how specific anti-malware solutions are particularly sensitive to certain obfuscation techniques, thus pointing out that it is not necessary to resort to extremely complex obfuscation strategy to obtain good evasion performances. We argue that an improved static analysis of the code, as well as some dynamic analysis, are required to improve robustness against obfuscated malware.

Our analysis also aimed at evaluating the easiness of deceiving anti-malware engines, with respect to false positives. To do so, we automatically injected strings belonging to malicious samples into a benign sample, thus creating an entire *fake malware* dataset that contains the strings of the Malgenome and Contagio datasets embedded into benign samples. Reported results showed that the majority of the anti-malware engines classifies the benign samples as malware, thus pointing out a deep weakness in their detection capabilities. Hence, if an attacker was able to make the anti-malware raise a lot of false-alerts, the user might be discouraged at using such system.

We can thus conclude, from our extensive evaluation over different obfuscation strategies and under different conditions, that anti-malware engines have significantly improved their resilience to obfuscation in comparison to the past years and, for a subset of them, evasion is much more difficult. However, for a number of anti-malware engines, and for a number of malware families, it is still possible to evade detection without resorting to complex obfuscation mechanisms. Some recent security reports clearly pointed out the rise of obfuscated malware that leverage on these weaknesses. So, additional effort is needed from anti-malware developers to take into account obfuscation mechanisms when designing anti-malware heuristics and signatures.

As soon as the vast majority of anti-malware detection systems improves its robustness against obfuscation techniques, it is likely that obfuscation will become more difficult to implement, and may end up to be too costly to be effective, as our experiments clearly pointed out.

Acknowledgments

This work is supported by the Regional Administration of Sardinia, Italy, within the project "Advanced and secure sharing of multimedia data over social networks in the future Internet" (CUP F71J11000690002). Davide Maiorca gratefully acknowledges Sardinia Regional Government for the financial support of his PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1.).

References

[1] F-Secure, Mobile Threat Report - Q1 2014, https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf (March 2014).

[2] Amazon app store, http://www.amazon.com/appstore.

[3] Samsung app store, http://apps.samsung.com.

[4] AV Comparatives, Cybercriminals infiltrate Android markets, http://www.av-comparatives.org/wp-content/uploads/2013/08/apkstores_investigation_2013.pdf.

[5] Dendroid malware can take over your camera, record audio, and sneak into Google Play, https://blog.lookout.com/blog/2014/03/06/dendroid/.

[6] An incredibly sneaky piece of malware has finally been pulled from Google Play, http://bgr.com/2014/06/18/google-play-store-android-malware-app/.

[7] Banking Malware Pulled From Google Play, http://securitywatch.pcmag.com/mobile-security/325324-banking-malware-pulled-from-google-play.

[8] Bad Certificate Management in Google Play Store, http://researchcenter.paloaltonetworks.com/2014/08/bad-certificate-management-google-play-store/.

[9] Bad Certificate Management in Google Play Store, https://bluebox.com/technical/android-fake-id-vulnerability/.

[10] X. Jiang, Security alert: New droidkungfu variants found in alternative chinese android markets, http://www.cs.ncsu.edu/faculty/jiang/DroidKungFu2/ (2011).

[11] Lookout Labs, Security alert: Malware found targeting custom roms (jsmshider), https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/ (2011).

[12] R. Unuchek, The most sophisticated android trojan, https://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan (June 2013).

[13] C. Ionescu, Obfuscating embedded malware on android (June 2012).

[14] R. Yu, Ginmaster: a case study in android malware, in: Virus Bulletin Conference, 2013.

[15] A. Apvrille, R. Nigam, Obfuscation in android malware, and how to fight back, https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation.

[16] Mario Ballano, Android malware, www.itu.int/ITU-D/eur/rf/cybersecurity/presentations/symantec-itu_mobile.pdf.

[17] E. Lafortune, Proguard, http://developer.android.com/tools/help/proguard.html.

[18] Saikoa, Dexguard, http://www.saikoa.com/dexguard.

[19] M. Zheng, P. P. C. Lee, J. C. S. Lui, Adam: An automatic and extensible platform to stress test android anti-virus systems, in: DIMVA - Detection of Intrusions and Malware, and Vulnerability Assessment - 9th Int. Conf., 2012, pp. 82–101.

[20] V. Rastogi, Y. Chen, X. Jiang, Droidchameleon: Evaluating android anti-malware against transformation attacks, in: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13, ACM, New York, NY, USA, 2013, pp. 329–334.

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

[22] Y. Zhou, X. Jiang, Android malware genome project, http://www.malgenomeproject.org/ (2012).

[23] M. Parkour, Contagio Mobile - Mobile Malware Mini Dump, http://contagiominidump.blogspot.com/.

[24] Cisco, Snort, http://www.snort.org.

[25] Baksmali, https://code.google.com/p/smali/.

[26] Android Open Source Project, Bytecode for the dalvik vm (2007).

[27] G. Nolan, Decompiling Android, Apress, 2012.

[28] Oracle, Java reflection api, http://docs.oracle.com/javase/tutorial/reflect/.

[29] Nihilus, Reversing dexguard 5.x, http://androidcracking.blogspot.de/2013/10/nihilus-reversing-dexguard-5x.html (October 2013).

[30] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Security and Privacy (SP), 2012 IEEE Symposium on, 2012, pp. 95–109.

[31] AV Comparatives, http://www.av-comparatives.org/.

[32] VirusTotal, https://www.virustotal.com.

[33] F. Maggi, A. Valdi, S. Zanero, Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors, in: Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), ACM, 2013.

[34] H. Huang, S. Zhu, P. Liu, D. Wu, A framework for evaluating mobile app repackaging detection algorithms, in: TRUST, 2013, pp. 169–186.

[35] M. Protsenko, T. Müller, Pandora applies non-deterministic obfuscation randomly to android, in: MALWARE, IEEE, 2013, pp. 59–67.

[36] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations (1997).

[37] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, K. Rieck, Drebin: Efficient and explainable detection of android malware in your pocket, in: Proc. of 17th Network and Distributed System Security Symposium (NDSS), 2014.

[38] Dasho, https://www.preemptive.com/products/dasho.

[39] Dexprotector, http://dexprotector.com/.

[40] Allatori, http://www.allatori.com/features/android-obfuscation.html.

[41] Apkfuscator, https://github.com/strazzere/APKfuscator (2013).

[42] Androguard, http://code.google.com/p/androguard/.

[43] Dedexer, http://dedexer.sourceforge.net/.

[44] A look at google bouncer, http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/.

[45] Anubis, https://anubis.iseclab.org/.

[46] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, S. Ioannidis, Rage against the virtual machine: Hindering dynamic analysis of android malware, in: Proceedings of the Seventh European Workshop on System Security, EuroSec '14, ACM, New York, NY, USA, 2014, pp. 5:1–5:6. doi:10.1145/2592791.2592796.
URL http://doi.acm.org/10.1145/2592791.2592796