Obfuscated Android Application Development
Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong

▶ To cite this version:
Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong. Obfuscated Android Application Development. CECC 2019 - Central European Cybersecurity Conference, Nov 2019, Munich, Germany. pp.1-6, 10.1145/3360664.3361144. hal-02305924

HAL Id: hal-02305924
https://hal-centralesupelec.archives-ouvertes.fr/hal-02305924
Submitted on 4 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Obfuscated Android Application Development

Pierre Graux
pierre.graux@inria.fr
CentraleSupélec, Inria, Univ Rennes,
CNRS, IRISA
Rennes, France

Jean-François Lalande
jean-francois.lalande@inria.fr
CentraleSupélec, Inria, Univ Rennes,
CNRS, IRISA
Rennes, France

Valérie Viet Triem Tong
valerie.viet_triem_tong@inria.fr
CentraleSupélec, Inria, Univ Rennes,
CNRS, IRISA
Rennes, France

ABSTRACT

Obfuscation techniques help developers to hide their code when distributing an Android application. The used techniques are linked to the features provided by the programming language but also with the way the application is executed. Using obfuscation is now a common practice and specialized companies sell tools or services for automatizing the manipulation of the source code. In this paper, we present how to develop obfuscated applications and how obfuscation technique usage is evolving in the wild. First, using advanced obfuscation techniques requires some advanced knowledge about the development of Android applications. We describe how to build such applications for helping researchers to generate samples of obfuscated applications for their own research. Second, the use of obfuscation techniques is evolving for both regular applications or malicious ones. We aim at measuring the development of these usages by studying application and malware samples and the artifacts that indicate the use of obfuscation techniques.

CCS CONCEPTS
- Security and privacy → Malware and its mitigation; Software reverse engineering.

KEYWORDS
obfuscation, mobile, application

ACM Reference Format:

1 INTRODUCTION

Obfuscation techniques help to protect an application from malicious users or competitors that want to gain knowledge about it. Understanding the internals of an application could be a legitimate need. For example, a security officer could be in charge of precisely understanding what is doing an application, before authorizing its installation in a company. On the other hand, malware developers want to protect the malicious code or some artifacts of the code (remote server URLs, ciphering keys,...). Whatever the purpose, whether legitimate or not, the source code or the compiled code is manipulated to slow down or to prevent at least its static analysis.

Due to the nature of Android applications, specific obfuscation techniques using the capabilities of the development framework have been created. These techniques can work on both the source code or the compiled bytecode, in addition to the possibility to execute native libraries. Indeed, an Android application is delivered to the end user using a specific format of the Java bytecode, interpreted by a virtual machine named Dalvik. Besides, the operating system can compile parts of the bytecode before or during the execution. This flexibility makes possible to develop obfuscation techniques, working on the bytecode directly, without access to the original source code. Using combinations of these techniques makes obfuscated applications difficult to reverse.

In this paper, we first briefly review the seminal works about generating obfuscated Java programs and later, obfuscated Android applications. Then, we propose in Section 3 to pedagogically describe development techniques for helping the reader to build obfuscated applications. These techniques are illustrated by snippets of code for the sake of clarity, even if the presented technique can be used directly on the bytecode. When developing new deobfuscation techniques, researchers can use these examples as unitary tests for improving their approach. Additionally, we discuss the difficulty to decide if an application has been obfuscated or not using the presented development technique. Indeed, finding artifacts of the use of a technique is often an easy task but deciding if the developer intent was to obfuscate parts of its code is a difficult problem. Finally, in Section 4 we present an overview of the use of these techniques on several application and malware datasets.

2 RELATED WORKS

Obfuscation techniques for Java programs have been firstly studied by Collberg et al. [3]. They distinguish the obfuscation of the control flow, the layout of the program and the manipulated data. All techniques are discussed in term of cost for the obfuscator (how it is easy to implement a transformation), of obfuscation robustness (the added time to reverse), of performance at execution time (the overhead). This seminal work helps to understand the transformations presented in this paper, such as field renaming or string encryption. Nevertheless, some transformations like reordering statements in basic blocks would be of no interest if the program is analyzed by automated tools or executed symbolically. Additionally, the developer could want to hide some assets, more than slowing down the reverse process. This explain why, when applied to Android system, developers would prefer to rely on transformations that hide the data more than complexify the flow.

1 Available at: https://gitlab.inria.fr/cidre-public/obfuscated-android-unit-examples
Table 1: Overview of our analysis methods

<table>
<thead>
<tr>
<th>Development technique</th>
<th>Analysis method</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>String encryption</td>
<td>usage ratio</td>
<td>yes (threshold)</td>
</tr>
<tr>
<td>Identifier renaming</td>
<td>usage ratio</td>
<td>yes (threshold)</td>
</tr>
<tr>
<td>Reflection</td>
<td>use</td>
<td>no</td>
</tr>
<tr>
<td>Dynamic code loading</td>
<td>use</td>
<td>no</td>
</tr>
<tr>
<td>Native method</td>
<td>use</td>
<td>no</td>
</tr>
<tr>
<td>Packer</td>
<td>artifacts</td>
<td>yes</td>
</tr>
</tbody>
</table>

As Android enables to escape the Dalvik virtual machine into native code, new possibilities of obfuscation have been designed. Rastogi et al. [8, 9] have worked on the design of new code obfuscation techniques and how to detect them. They have split the obfuscation techniques in two categories: the ones that can be defeated by static analysis tools or the ones that cannot. Only the use of reflection and bytecode encryption is presented as circumventing a static analysis as the static analyzer cannot even parse the code. Nevertheless, techniques used by packers, as shown in this paper, defeat easily the static analysis of the bytecode.

Recently, Dong et al. [4] contributed on the problem of detection of obfuscation techniques in Android applications. Their approach relies on two techniques: first, they parse the bytecode for collecting the used strings or identifiers (class, method and field names); second, if needed, they build on the collected data vectors of features using n-grams for preparing a classification phase. Then, they detect an obfuscation by classifying the vectors with an SVM classifier. This requires to train the classifier on labeled datasets which is a constraint we want to avoid. The obtained results on applications extracted from markets and malware repositories have been validated on manually analyzed datasets of goodware and malware. Nevertheless, the used datasets only focus on years 2016 and 2017 and do not use datasets from the literature. Finally, as they rely on a SVM classifier, it is difficult to express the reasons that make a sample classified as obfuscated or not. The detection rule, contained in the SVM classifier, is not expressed using real life elements (such as “this specific API is used”), but is a condition on vector values.

In this paper, we propose to enhance the results of Dong et al. [4] by proposing the following new contributions. First, we describe carefully how to create an application using development techniques that can be used for obfuscation. These techniques are listed in the left column of Table 1 and described from the most simple ones to the most advanced ones. Second, we provide static analysis methods for detecting the use of these development techniques. We either detect the use or the usage ratio if this technique can be applied on multiple parts of the code (listed in the middle column of Table 1). When possible, the analysis method is extended as a detection decision, as shown in the right column of Table 1. Finally, we analyze both goodware and malware datasets of the literature and give a global view of development techniques over the years 2008–2018, a time period five times longer than Dong et al.

3 BUILDING OBFUSCATED APPLICATIONS

To have a simple explanation, this section relies on a running example of code that could be found in malware sample. This code is typical of the code used by malware to communicate with its command and control server (C&C). This code is presented in Listing 1. This code has three sensitive assets: the value of two constants representing the url and the port of the C&C server and the call to the method send() of the DatagramSocket class used to communicate with the C&C server.

3.1 String encryption

3.1.1 Description. The string encryption is a technique that consists in replacing string constants by an encrypted version [8]. Before using the constant, the code has to decrypt it. When reverse engineering an APK, all the constant strings are available. Thus, by encrypting them, the developer forces the analyst to understand the decryption code before accessing the original strings. Additionally, the encryption key can be hidden in the code, or downloaded dynamically, which increases reverse engineering efforts. As we can see in Listing 2, the C&C url is no more directly available because of the use of the AES256 ciphersing algorithm.

3.1.2 Detection. To detect the usage of string encryption, a naive approach is to compute the entropy of strings. Indeed, encryption algorithms such as AES, DES, RC4, output bytes with a high entropy close to the random distribution entropy. Thus, we propose to extract all strings of an application (excluding field, method and class names) and compute their entropy. The proposed detection method consists in counting the number of strings m with an entropy greater than an entropy threshold. Then, we say that an application has been obfuscated for n strings if m is greater than n. We discuss the choice of the threshold and we show the effect of choosing n = 1, 10 or 100 in Section 4.

Listing 1: Unobfuscated malicious code

```java
1 class CommandAndControl {
2     public String url = "cc.url";
3     public int port = 4242;
4     public void sendCommand(byte[] buf, int length) {
5         DatagramPacket dp = new DatagramPacket(buf, length,
6             InetAddress.getByName(this.url),
7             this.port);
8         new DatagramSocket().send(dp);
9     }
10 }
```

Listing 2: String encrypted version

```java
1 class AES256 {
2     static public String aes256(String str) {
3         String key = "<key>"... 
4     }
5 class CommandAndControl {
6     public String url = "<ciphered url>"; // "cc.url"
7     public int port = 4242;
8     public void sendCommand(byte[] buf, int length) {
9         DatagramPacket dp = new DatagramPacket(buf, length,
10             InetAddress.getByName(AES256.aes256(this.url)),
11             this.port);
12     new DatagramSocket().send(dp);
13 }
14 }
```

https://source.android.com/devices/tech/dalvik/dex-format
We remove all the potential small words of size lesser than 4, that we retrieve all the identifiers and we split them by underscore and we prefer to rely on a detection method not based on a precise word list. The proposed detection method consists in comparing words to build their identifiers (either CamelCase or snake_case). To detect the usage of identifier renaming, we propose to automate this renaming process.

To detect the usage of identifier renaming, we propose to automate this renaming process.

A more advanced detection technique have been used by Dong et al. [4]. It uses 3-grams to traverse the identifier names and an SVM classifier to recognize encrypted identifiers. This technique requires to train the classifier on a manually labeled dataset.

These detection techniques can be defeated by the use of encryption algorithms that does not modify the entropy, for example ROT13. Additionally, strings that are converted in integer arrays escape the detection.

3.2 Identifier renaming

3.2.1 Description. Similarly to the string encryption technique, the identifier renaming technique consists in replacing code identifiers by obfuscated ones. As the constant strings, the identifiers (package, class, method and field names) are available directly in the APK. By replacing the identifiers by meaningless ones, the developer prevents the analyst to guess the purpose of an identifier. Moreover it can confuse him by using identifiers hard to distinguish or to remember, as shown in Listing 3. Some tools such as ProGuard, propose to automate this renaming process.

3.2.2 Detection. To detect the usage of identifier renaming, we have implemented a naive approach based on word lists. This heuristic relies on the fact that usually developers use composition of words to build their identifiers (either CamelCase or snake_case). We retrieve all the identifiers and we split them by underscore and uppercase in order to get the constitutive words of the identifier. We remove all the potential small words of size lesser than 4, that may be abbreviations. Then, we check the ratio $r$ of the number of non-meaningful identifiers over the total, by searching them in a word list. The proposed detection method consists in comparing $r$ with a chosen threshold. We discuss the choice of the threshold in Section 4.

Similarly to string encryption method, Dong et al. have used 3-grams and an SVM classifier. Although they obtain good results [4], the learning phase uses applications obfuscated by Proguard and DashO, which implies that the detection method is efficient with these two tools. Using an unknown obfuscator renaming identifiers may be not detected by the learning machine. For these reasons, we prefer to rely on a detection method not based on a precise obfuscation tool.

3.3 Reflection

3.3.1 Description. The reflection usage technique aims at hiding the methods and fields that the code is calling [8]. Developers use an API to access and modify at runtime internal object entities such as fields of an object. For obfuscation purpose, using reflection removes the direct references to the type or the method names that are used in the bytecode. For example in Listing 4, the methods getByName and send will not appear in the bytecode of the sendCommand method. Coupled with the string encryption technique, the use of reflection prevents the APK to be easily statically analyzed.

3.3.2 Detection. Because reflection mechanisms are given by the Java API java.lang.reflect and thus cannot be renamed, its detection is trivial [4]: one only has to search for references to methods belonging to this package.

It has to be noted that reflection can be used for non-obfuscation purpose. Thus, it is non trivial to distinguish reflection usages that are made for obfuscation purpose and those that are not. Dong et al. [4] have developed a more advanced technique that analyzes instructions of the bytecode to recover the name of the class on which is invoked a reflection method, but this extra information is not sufficient to determine if the goal is to obfuscate.

3.4 Dynamic code loading

3.4.1 Description. Dynamic code loading can be used for loading dynamically some classes when not available at compile time or when building a distributed application loading remote components from a server. In addition, the bytecode can be ciphered and deciphered before loading it [8]. DexFile and PathClassLoader provide helper classes for mapping a dex file into the memory and instanciate new objects. The updated snippet of code is given in Listing 5: after downloading a jar file, the send() method is invoked using reflection.

Substituting totally the class loader at runtime is also possible [5]. Nevertheless, it requires to access and update the private attribute $mClassLoader$ of the class $LoadedApk$ of the application.

3.4.2 Detection. Because dynamic code loading mechanisms are given by specific classes of the Android API, its detection is trivial [4] we only need to search for occurrences of all class loader classes. Once again, this detection does not distinguish between dynamic code loading used for obfuscation purpose and the others.

---

**Listing 3: Identifier renaming version**

```java
1 class ____ {  
2   public String ____ = "cc.url";
3   public int ____ = 4242;
4   public void ____ (byte[] ____, int _____) {  
5       DatagramPacket _______ = new DatagramPacket_____,  
6       InetAddress.getByName(this.____),  
7       new DatagramSocket().send(_______);
8       this.___;
9   }
10 }
```

**Listing 4: Reflection version**

```java
1 class CommandAndControl {  
2   public String url = "cc.url";
3   public int port = 4242;
4   public void sendCommand(byte[] buf, int length) {  
5     InetAddress addr = InetAddress.class
6       .getDeclaredMethod("getByName", String.class)  
7       .invoke(null, this.url);
8     DatagramPacket dp = new DatagramPacket(buf, length,  
9           this.port);
10    DatagramSocket.class.getDeclaredMethod("send",  
11       DatagramPacket.class)  
12    .invoke(new DatagramSocket(), dp);
13 }
```
which protects its internal structure representing the loaded APK.

Numerous packing services cause the bytecode is constrained by the Dalvik virtual machine. Usually, the unpacking routine is developed using native code beforehand in the bytecode and is in charge of decrypting and loading the original bytecode. Thus, the bytecode is only available at runtime. Using the bytecode and is in charge of decrypting and loading the original bytecode. Thus, the bytecode is only available at runtime.

A packer is a tool that encrypts the bytecode of an APK, stores it in a resource file and removes the original bytecode. For example, an analyst that searches for the usage of libc socket, as shown in Listing 6.

3.5 Native method

3.5.1 Description. For performance purpose, Android gives to the developers the ability to directly write methods in C++ which are then compiled in assembly. Such methods are called native. Comparatively to the bytecode, the assembly is difficult to decompile. Thus, it can be used to circumvent tools that only work on the bytecode. For example, an analyzer that searches for the usage of network package methods, can miss the usage of libc socket, as shown in Listing 6.

3.5.2 Detection. The detection of native methods is easy because they have to be declared as native in the bytecode and their implementation are located in a specific compiled library. However it is difficult to determine if native methods are used for obfuscation purpose or for a legitimate use, for example graphical routines using OpenGL ES.

3.6 Packer

3.6.1 Description. A packer is a tool that encrypts the bytecode of an APK, stores it in a resource file and removes the original bytecode. Finally, it adds an unpacking routine which is called before the bytecode and is in charge of decrypting and loading the original bytecode. Thus, the bytecode is only available at runtime. Usually, the unpacking routine is developed using native code because the bytecode is constrained by the Dalvik virtual machine which protects its internal structure representing the loaded APK against modification. Numerous packing services propose to pack

```
Listing 5: Dynamic code loading version

1 void invokeDynamically() {
2    // Loads the implementation of C&C */
3    PathClassLoader pcl =
4        new PathClassLoader(file.getPath(), null);
5    Class<?> clazz =
6        pcl.loadClass("CommandAndControl", this);
7    Log.i("CL", "Loaded class from dex: "+ clazz);
8    Constructor<?> c = clazz.getConstructor();
9    Object o = c.newInstance();
10   Method m = clazz.getMethod("send",
11       byte[].class, int.class);
12   m.invoke(o, ...); // CommandAndControl.send(...)
13 }
```

```
Listing 6: Native version

3.6.1 Description. A packer is a tool that encrypts the bytecode of an APK, stores it in a resource file and removes the original bytecode. Finally, it adds an unpacking routine which is called before
```

```
Listing 7: Packer replacing the bytecode of a method

1 void unpack_method(jclass cls, jstring methodName,
2     jstring methodSignature) {  // Get the method ID that will be overwritten */
3    /* In fact, in Android runtime, this ID is a pointer to an internal class named ArtMethod */
4    void* art_method = (void*) env->GetMethodID(cls,
5        methodName, methodSignature);
6    /* Get the dex_code_item_offset_ field of the ArtMethod class */
7    The hardcoded offset (8) has been retrieved using objdump on the compiled runtime. This offset could change from one Android version to another, */
8    unsigned int code_item_offset_ = *(unsigned int*) ((
9        char*)art_method + 8);
10   /* Find the location of the DEX in the memory using /proc/self/maps. To found the DEX area, search for the .dex file */
11   void* dex_file_location = getDexFileLocation();
12   /* Get the new bytecode from a PNG file */
13   const void* mmaped_file_location = GetXoredApk();
14   /* Retrieve the CODE_ITEM structure using its offset from the beginning of the DEX. CODE_ITEM is a DEX structure and is documented */
15   void* code_item_ = (void*)((char*)dex_file_location +
16       code_item_offset_);
17   /* Retrieve the bytecode size and the bytecode address by navigating in the CODE_ITEM structure */
18   unsigned int APK_insns_size_in_code_units_ = *(unsigned int*)
19       ((char*)code_item_ + 12);
20   void* APK_insns_ = (void*)((char*)code_item_ + 16);
21   /* Do the same for the PNG (xored APK). Recomputing code_item_offset is not needed because the offsets in the APK and in the xored APK are the same */
22   code_item_ = (void*)((char*)mmaped_file_location +
23       code_item_offset_);
24   unsigned int PNG_insns_size_in_code_units_ = *(unsigned int*)
25       ((char*)code_item_ + 12);
26   void* PNG_insns_ = (void*)((char*)code_item_ + 16);
27   /* Set the method instructions writable */
28   void* base_addr = (void*)((char*)APK_insns_ -
29       APK_insns_size_in_code_units_ +
30       mprotect(base_addr, size_t((char*)APK_insns_ +
31       APK_insns_size_in_code_units_ + 2 - (char*)base_addr), PROT_READ|PROT_WRITE|PROT_EXEC);
32   /* Un-xor and copy all new instructions. 0x42 is the xor key */
33   unsigned int i;
34   for(i=0 ; i < APK_insns_size_in_code_units_ ; i++) {
35       *(char*)APK_insns_ + 2*i  = *(char*)PNG_insns_ + 2*i ^ 0x42;
36       *(char*)APK_insns_ + 2*i + 1 = *(char*)PNG_insns_ + 2*i + 1 ^ 0x42;
37   }
38 }
```

application sent by customers. However, packer’s internals are not documented. Thus, we give in the following more insight on how to implement a packer and we focus on the native code that unpack and replace parts of the bytecode. As dynamic code loading can be easily detected by a dynamic tool hooking the relevant methods, packers use native code to
change the Dalvik virtual machine internal structures representing the APK and the mapped bytecode of the memory. This principle can be exploited at different levels. Packers can either populate the bytecode for one method, for the whole class, or for the whole APK.

As an example, we present a packer that populate the bytecode of a method [5]. This application locates the packed bytecode from the file /proc/self/maps where the APK file is located in the memory. Then, it fully overwrites its bytecode. To locate the bytecode that needs to be re-written, the code navigates in internal structures of the Android runtime. Finally, to ensure to be run before any other bytecode, the unpacking routine can be implemented in a class Application and set in the Manifest under the android:name parameter of the application manifest block. The main steps of the native code that unpacks and replaces the bytecode is given in Listing 7.

3.6.2 Detection. To detect the presence of packer we use APKiD [7], a tool that searches static artifacts of known packers in APKs. However, an “home-made” packer would not be detected. Detecting unknown packer remain an open research problem [6, 10, 13].

4 OBfuscATION USAGE IN DATASETS

In this section, we discuss the use of development techniques, used for obfuscation or not, for goodware and malware datasets.

4.1 Used datasets

The four datasets used in our experiments are: two well known but old datasets (AMD dataset [12] and the Drebin dataset [2]), one more recent (GM19 dataset [11]) and one large (AndroZoo [1]). Drebin is the oldest one: it contains 5560 samples from 2008 to 2012. AMD contains 24552 samples from 2010 to 2016. GM19 contains two balanced sets of 5000 goodware and 5000 malware with an homogeneous repartition of dates (2015-2018) and APK size. Finally, we built a custom large dataset by picking randomly 9,041 applications from AndroZoo in order to cover the largest time interval.

4.2 Packer, native, DCL, reflection

Table 2 compares the detection rates for the different techniques presented in Section 3. When a packer is detected, the tests for native method, dynamic code loading and reflection are not run because they are static techniques and would end up analyzing the unpacker code instead of the application code itself. The percentages associated to these three techniques are computed for the non-packed applications only.

Table 2: Obfuscation detection for various datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Total</th>
<th>Packer</th>
<th>Native</th>
<th>DCL</th>
<th>Reflection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0,06%</td>
<td>25,58%</td>
<td>91,82%</td>
<td>95,68%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10,86%</td>
<td>57,62%</td>
<td>90,38%</td>
<td>94,33%</td>
</tr>
<tr>
<td>AMD [12]</td>
<td>24552</td>
<td>31</td>
<td>5146</td>
<td>13267</td>
<td>19184</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,13%</td>
<td>21,52%</td>
<td>63,12%</td>
<td>79,31%</td>
</tr>
<tr>
<td>Drebin [2]</td>
<td>5560</td>
<td>0</td>
<td>1051</td>
<td>1449</td>
<td>3066</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0%</td>
<td>19,07%</td>
<td>26,29%</td>
<td>55,82%</td>
</tr>
</tbody>
</table>

When comparing the results of the recent dataset MAL with the old AMD and Drebin datasets, we globally see an increase of native methods, dynamic code loading and reflection usage. This is confirmed by the experiment on the subset of Androzoo for which we represented the detection rate over the years 2008 to 2018 in Figure 1. In this figure, we also note an increase of the packer usage after 2014. The reader should note that the years 2008 to 2010 contain few samples analyzed (less than 164) which make the curves less precise for these dates. In particular, the year 2008 is based on 61 samples only and results seem biased.

When comparing goodware (GOOD dataset) and malware (MAL dataset), it is clear that native methods are more frequently used in malware samples. On the contrary, dynamic code loading and reflection does not discriminate a goodware from a malware. Indeed the usage of such development technique is not by itself an obfuscation attempt. For example, Google libraries, that are embedded in goodware APK file, contains reflection calls and class loading mechanisms even if these libraries do not use any obfuscation techniques (they are open-source).

4.3 String encryption and identifier renaming

For studying globally the identifier renaming, we represented in Figure 2, for each dataset, the percentage of APKs detected for every possible threshold. An application is said to use identifier renaming if the number of identifiers out of the dictionary divided by the total number of identifier is greater than the threshold. We observe that the curves decrease slowly until reaching a ratio threshold of 0.5. Thus, almost all applications have more than 50% of identifiers out of the dictionary which can be explained by our dictionary nature: it only contains natural language and no technical words of computer science domain. We also observe a slow down of the decreasing of the curve around the ratio threshold of 0.8. That means that less than 13% of applications have their identifiers out of our dictionary, and thus are probably obfuscated. Thus, we propose a detection threshold of 0.8 for deciding if an application has been obfuscated by renaming identifiers.
scored are really true positives. Thus, we took the 5 top most detected malware of each dataset and reversed them manually. We observed 40\% of true positives (60\% of false positive) for string encryption and 100\% of true positive (0\% of false positive) for identifier renaming. The observed false positive are due to a legitimate usage of high entropy strings such as base64 encoded strings or fields containing the full english alphabet.

5 CONCLUSION

This paper has presented obfuscation techniques used by applications developers. We concentrated on showing how to develop an obfuscated application. Experiments show the increasing usage of these techniques. We also proposed simple detection techniques for obfuscation method that are simple to develop. They help to quantify the usage of each obfuscation technique. Nevertheless, choosing the detection thresholds that have a high true positive rate and low false negative rate remains a problem to investigate.

REFERENCES