2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)

Abusing Android Runtime for Application Obfuscation

Pierre Graux, Jean-Francois Lalande, Pierre Wilke, Valérie Viet Triem Tong
CentraleSuplec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
firstname.lastname@inria.fr

***Abstract*—Studying Android obfuscation techniques is an essential task for understanding and analyzing malicious applications. Obfuscation techniques have already been extensively studied for market applications but never for pre-compiled applications used in smartphone firmwares. In this paper, we describe two new obfuscation techniques that take advantage of the duality between assembly and Dalvik bytecode and, as far as we know, have never been described before. We also propose detection methods for these obfuscation techniques. We apply them to vendor firmwares and market applications in order to evaluate their usage in the wild. We found that even if they do not seem to be already used in the wild, they are fully practical.**

***Index Terms*—Obfuscation, Android, Assembly**

1. Introduction

The tremendous number of malicious Android applications has created a crucial need for analyzing Android obfuscation techniques. Although most malicious applications are gathered on the Google Play market or other third-party markets, firmwares can also contain malicious applications. For example, in 2017, Kryptowire[1] discovered Android firmwares that transmit private information to third-party servers, such as the full content of the user's text messages. In such cases, the malicious application is not installed directly by the user but through the Android update mechanism, or is already on the device when sold.

The introduction of malicious code into firmwares gives the attacker a specific opportunity to use assembly-based obfuscation techniques. These techniques could use assembly code to perform actions that work only on the targeted smartphone. Proposing such obfuscation techniques has never been explored yet. Indeed, state-of-the-art obfuscations have usually been studied only at the application level.

In this article, we propose two new obfuscation techniques that use assembly code to hide the malicious intents of applications and we present methods to detect the use of such techniques. The first one is called Bytecode-Free OAT (BFO) in reference to the Android file format OAT. It consists in removing all the bytecode from an application while leaving its assembly unchanged. It is only applicable on pre-compiled applications, that is, only in firmwares. The second one is called Direct Heap Access (DHA) and consists in using assembly code to access the Android memory layout in order to bypass the interface between bytecode and native code and therefore thwart most analysis tools. This technique can be used on both market and firmware applications. In order to evaluate if these techniques are already actively used in the wild, we propose detection techniques that we apply to application and firmware datasets.

The contributions of this study are the following:

- we describe a new obfuscation technique, Bytecode-Free OAT, based on the removal of the Dalvik bytecode, therefore leaving only the compiled assembly;
- we describe a new obfuscation technique, Direct Heap Access, based on the bypassing of the interface between bytecode and native code;
- we present methods to detect the use of these new techniques;
- we evaluate the usage of these techniques in smartphone firmwares and, for Direct Heap Access only, in application datasets.

The rest of the article is organised as follows. Section 2 describes the necessary background on the Android system. Sections 3 and 4 present, respectively, BFO and DHA and their corresponding detection techniques. Section 5 shows the results obtained when using these detection techniques in the wild. Section 6 refers to works related to Android assembly obfuscation. Finally, Section 7 concludes the article.

2. Android runtime overview

An Android application is distributed as an APK file, which is an archive file containing the resources, code and metadata of the application. The code can be either Dalvik bytecode, stored in DEX files (Dalvik EXecutable), or assembly code, stored in shared libraries (`.so` files). We will respectively name them bytecode and native code in the remaining of this article. Usually, bytecode is compiled from Java or Kotlin source code, and native code is compiled from C/C++. In order to access bytecode objects, which are stored on the heap, Android provides the Java Native Interface (JNI). This interface allows native code to request the runtime to perform operations such as retrieving or setting an object field, calling an object method or even creating a new instance of a class. The main purposes of JNI and native code is to improve the performance of applications.

Moreover, since Android 7.0 (Nougat, 2016), the Android runtime (ART) is able to compile bytecode methods into assembly. The compilation is performed on the

1. https://www.kryptowire.com/kryptowire-discovers-mobile-phone-firmware-transmitted-personally-identifiable-information-pii-without-user-consent-disclosure/

DOI 10.1109/EuroS&PW51379.2020.00088

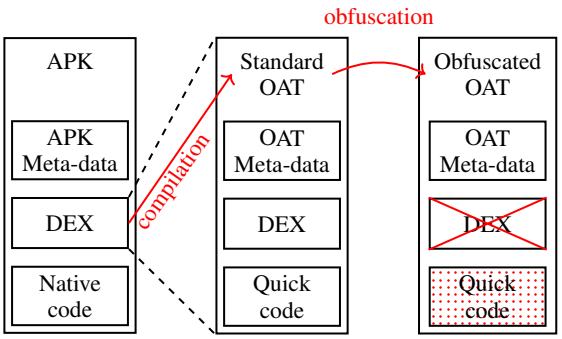

Figure 1: Bytecode-Free OAT (BFO) technique

smartphone itself, which allows the compiler to use optimizations tuned for a specific architecture. This greatly improves the performance of the application. When compiled, Dalvik bytecode is stored on the phone using a new file format called OAT[2]. This file contains the originally compiled DEX file together with the resulting assembly code. For the sake of clarity and according to the Android source code[3], the assembly code resulting from the bytecode compilation will be named *quick code* while the assembly code resulting of the C/C++ compilation will be named *native code*. When executing the application, the Android runtime runs the quick code. If, and only if, the quick code is not available, it runs the bytecode[4].

3. Bytecode-Free OAT

This section presents Bytecode-Free OAT (BFO), an obfuscation technique that takes advantage of the OAT file format as represented in Figure 1. When executing an application on Android, if some quick code is available, it is always executed, regardless of whether bytecode is present or not. So, an attacker could tamper with the bytecode without modifying the executed quick code. Thereby, the application behavior is not changed but the analysis of the bytecode would be erroneous since it is not performed on the code that is actually run. Quick code is only available in OAT files which are created on the smartphone itself and cannot be distributed. Thus, BFO cannot be used on markets such as Google Play. Instead, this technique would be particularly well suited for firmware vendors because these companies provide their applications already pre-compiled for a specific phone model.

Depending on how the bytecode is tampered with, the BFO technique can be divided in three different subtechniques described in the next three following sections: removing, replacing or modifying the bytecode. These techniques are classified according to three criteria: their robustness, their stealthiness and the possibility to automate them. Since assembly code works at a lower abstraction level than the bytecode, we consider that analyzing bytecode is simpler than analyzing assembly. Consequently, we consider that an obfuscation technique is more robust than another if it requires to analyze more assembly code. On the other hand, we consider that an obfuscation technique is stealthier than an other if the difference between the behavior described by the bytecode and the one observed is smaller: analysts only look at the assembly code if the result of the bytecode analysis seems incorrect with respect to the behavior of the application.

2. We could not find an official definition for this acronym.
3. AOSP source code: https://source.android.com/
4. The bytecode can be either interpreted or Just In Time compiled.

3.1. Removing the bytecode

The first variant of this BFO technique consists in removing or "nopping" the bytecode. This means replacing the bytecode by No-OPeration (NOP) instructions or, by extension, by instructions that do not have any special effect. Removing the bytecode is allowed by the OAT file format in order to represent `abstract` methods. The quick code, which is always executed regardless of the bytecode, is not modified, in order to preserve the application behavior.

This technique perfectly fools bytecode analysis tools since the information on which they perform their analyses is deleted. Instead, the reverse engineering of the application needs to be done directly on the assembly code. Additionally, this technique is easily automatable since the modifications applied to the bytecode are the same for all applications and do not depend on the bytecode.

However, using this technique is not stealthy since bytecode analysis cannot give any result if there is no bytecode provided at all. Moreover, the removal of the bytecode can be automatically detected if an application contains a method that has quick code but no bytecode. Results of this simple detection algorithm is presented in Section 5.1.1. For the "nop" version, the detection can be achieved using statistical properties such as entropy. Since nopping code consists in rewriting it using always the same pattern, it lowers the entropy of the code. By using an entropy threshold, the nopping can be detected. Section 5.1.2 discusses the results obtained with this detection technique.

3.2. Replacing the bytecode

As previously stated, removing or nopping the bytecode is not stealthy. To counter this drawback, the bytecode can be replaced instead of removed or nopped: the bytecode of a sensitive method, *e.g.* a method that checks the PIN code, can be replaced by a benign method, *e.g.* a hello world. Thus, the robustness of the obfuscation technique is kept while its stealthiness is improved: the bytecode still does not give any information about the real behavior of the application and analysis tools generate wrong results since they have bytecode to work on.

The automation of this technique is technically possible, but not trivial. The bytecode cannot be replaced by repetitive patterns of bytecode because this would be easily detectable as previously stated in Section 3.1. The automation can neither generate random patterns because this would result in incorrect bytecode, which is also easily detectable by a bytecode verifier. Thus, automating this technique requires to generate random valid bytecode, that is bytecode which respects, among other criteria, the signature of the replaced method. While it is still doable, it requires some engineering work, and is left as future work.

If correctly implemented, detecting this techique consists in detecting if a given assembly code is the result of the compilation of a given bytecode. This can be done by compiling the bytecode and then comparing the result of this compilation to the given assembly. Section 5.1.3 shows results obtained with this detection technique. While detecting automatically this technique is a challenging task, a manual investigation can easily detect bytecode replacement. The executed assembly code completely differs from the analyzed bytecode, which makes the results obtained by a bytecode analysis incoherent and thus can quickly hold the analyst's attention.

3.3. Modifying the bytecode

Finally, if the stealthiness of the code is considered more important than the robustness, a third BFO technique can be used. This techniques consists in smoothly modifying the bytecode. Instead of completely modifying the bytecode behavior, only a few instructions that are chosen very carefully are minutely modified.

For example, if someone wants to protect a code that contains a CRC check of incoming network packets, obfuscating the creation of the CRC table would be a typical goal. The bytecode corresponding to such a method is presented in Listing 1. This bytecode has been obtained by compiling an application containing CRC computations and inspecting the resulting DEX file. Line 8, the bytecode initializes the polynomial that is used to compute the table. If only this line is modified, as shown in Listing 2, the bytecode analysis of the application does not raise any alarm: the results is coherent with the behavior of the application.

However, this technique presents two main drawbacks. First, it is less robust. Even if the bytecode differs from the assembly, it still gives a lot of insight about what is the behavior of the application. Second, it is not automatable. Indeed, modifications that are made to the bytecode require a very good knowledge about the obfuscated bytecode.

```
1  1200           | const/4 v0, #int 0
2  1301 0800      | const/16 v1, #int 8
3  3510 1400      | if-ge v0, v1, 14
4  dd01 0401      | and-int/lit8 v1, v4, #int 1
5  1212           | const/4 v2, #int 1
6  3321 0a00      | if-ne v1, v2, 11
7  e201 0401      | ushr-int/lit8 v1, v4, #int 1
8  1402 2083 b8ed | const v2, #edb88320
9  9704 0102      | xor-int v4, v1, v2
10 2803           | goto 12
11 e204 0401      | ushr-int/lit8 v4, v4, #int 1
12 d800 0001      | add-int/lit8 v0, v0, #int 1
13 28eb           | goto 02
14 1500 00ff      | const/high16 v0, #int -16777216
15 b740           | xor-int/2addr v0, v4
16 0f00           | return v0
```

Listing (1) CRC32 bytecode

```
1  1200           | const/4 v0, #int 0
2  1301 0800      | const/16 v1, #int 8
3  3510 1400      | if-ge v0, v1, 14
4  dd01 0401      | and-int/lit8 v1, v4, #int 1
5  1212           | const/4 v2, #int 1
6  3321 0a00      | if-ne v1, v2, 11
7  e201 0401      | ushr-int/lit8 v1, v4, #int 1
8  1402 2ed8 31eb | const v2, #eb31d82e
9  9704 0102      | xor-int v4, v1, v2
10 2803           | goto 12
11 e204 0401      | ushr-int/lit8 v4, v4, #int 1
12 d800 0001      | add-int/lit8 v0, v0, #int 1
13 28eb           | goto 02
14 1500 00ff      | const/high16 v0, #int -16777216
15 b740           | xor-int/2addr v0, v4
16 0f00           | return v0
```

Listing (2) Modified CRC32 bytecode

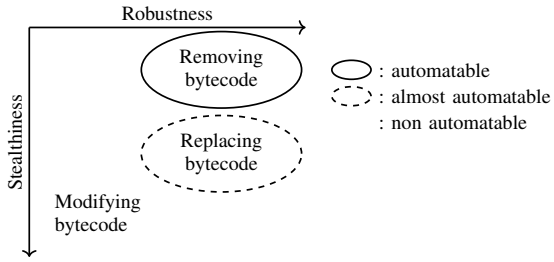

Figure 3: Classification of BFO sub-techniques

3.4. BFO sub-techniques comparison

The three BFO sub-techniques previously described are classified in Figure 3 according to their robustness, their stealthiness and the possibility to automate them. Removing the bytecode is the one that is the easiest to automate. However it is the least stealthy. Replacing the bytecode can be viewed as an improvement of simply removing, since it improves the stealthiness while not reducing the robustness. Nevertheless, its process is harder to automate. Finally, modifying the bytecode is the stealthiest sub-technique but reduces robustness of the obfuscation and requires manual editing.

4. Direct Heap Access

This section presents Direct Heap Access (DHA), a new obfuscation technique that consists in using native code to modify Java object fields directly on the heap without relying on bytecode or runtime functionalities. Indeed, as stated in Section 3.3, obfuscation techniques can consist in stealthily modifying values of carefully chosen fields. Modifying Java fields in native code is classically made by relying on the Java Native Interface (JNI). This interface contains methods that allow the native code to access the heap, where Java objects are stored. However, this interface is well known: analysis tools are able to setup hooks in this interface in order to retrieve the behavior of the assembly part of an application and to model how native code modifies the Java fields [1]–[3].

However, native code can modify Java fields without JNI by directly modifying their value. This allows to bypass state-of-the-art tools and is the purpose of DHA which is represented in Figure 4. The Dalvik virtual machine does not give any guarantee about how fields are stored in the heap. Consequently, directly reading or writing the heap without using JNI is not straightforward. In the following section, we provide three ways to implement DHA.

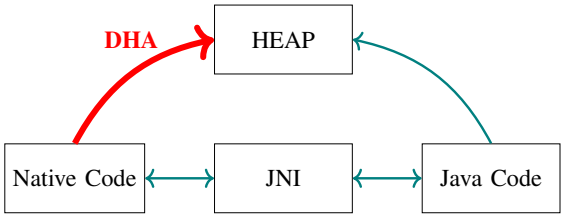

Figure 4: Direct Heap Access (DHA) technique

```
1 extern "C" JNIEXPORT void JNICALL bytebuffer(JNIEnv
      ↪ *env, jobject thisObj, jobject buf) {
2     void* addr = env->GetDirectBufferAddress(buf);
3 }
```

Listing 3: DHA using `DirectByteBuffer`

4.1. DHA implementation

The three DHA implementations given in this section are ordered increasingly by the knowledge required about Android runtime internals to implement them. The first implementation provided, in Section 4.1.1, describes a solution based on a legitimate use of `DirectByteBuffer`, a specific class provided by Android. Section 4.1.2 gives a naive way of doing a DHA by scanning the whole heap. Finally, Section 4.1.3 gives an advanced implementation which is able to navigate through the internal structures of the Dalvik virtual machine. Each implementation is shown using the example presented in Section 3.3. In this example, the obfuscation aims at modifying the value of the polynomial used to compute a CRC table.

4.1.1. Legitimate implementation: DirectByteBuffer. Implementing a legitimate DHA is facilitated by the Java class `ByteBuffer` which provides a way to allocate a buffer directly accessible by the native code. This buffer, named `DirectByteBuffer`, has a field, `address`, that localizes the bytes in the memory heap. It is created using the `allocateDirect` method from the `ByteBuffer` class. `DirectByteBuffer` has been made for native optimization purposes.

However the `address` field is not visible. That is why JNI provides `GetDirectBufferAddress` to directly access it, as shown in Listing 3. To avoid using JNI, which is the goal of DHA, this field can be retrieved using reflection. This is done in Listing 4. Using the obtained address, native code can directly access the content of the `DirectByteBuffer`. In any case, the native code needs to receive or retrieve the byte buffer address, which can be detected by state-of-the-art tools [2]. Thus, `DirectByteBuffer` does not fulfill the obfuscation goal that is realizing a stealthy access.

4.1.2. Naive implementation: memory lookup. Native code can avoid the need of receiving the address of a field by looking into the memory for the field value. Indeed, if the field has a specific unique value, such as `0xeb31d82e` in the CRC example, the native code can scan the memory to retrieve its location. Listing 5 shows this process by reading the special `/proc/self/maps` file. This file contains the memory mapping of the process that reads it, including the memory area named "dalvik-main space" which is the one that stores the fields. By searching the obfuscated field value inside this area, its address can be retrieved.

```
1 Field field;
2 field = Buffer.class.getDeclaredField("address");
3 field.setAccessible(true);
4 long addr = (long) field.get(directBytebuffer);
```

Listing 4: Retrieving `DirectByteBuffer` address without JNI

```
1   #define SEARCHED_VALUE 0xeb31d82e
2 extern "C" JNIEXPORT void JNICALL memLookup(JNIEnv *
      ↪ env, jobject thisObj) {
3   FILE *file = fopen("/proc/self/maps", "r");
4   if (file == NULL) return;
5   char *line = NULL;
6   size_t n = 0;
7   while (getline(&line, &n, file) > 0) {
8     char *path = strchr(line, '/');
9     if (!path) continue;
10    if (strcmp(path, "/dev/ashmem/dalvik-main space\
      ↪ n")!=0 && strcmp(path, "/dev/ashmem/dalvik-
      ↪ main space (deleted)\n")!=0) continue;
11    unsigned long vm_start, vm_end;
12    char r, w, x, s;
13    if (sscanf(line, "%lx-%lx %c%c%c%c", &vm_start,
      ↪ &vm_end, &r, &w, &x, &s) < 6)
14      continue;
15    if (r != 'r' || w == 'w') continue;
16    for(unsigned long i=0; i < vm_end-vm_start-
      ↪ sizeof(unsigned long) ; i++)
17      if(*(unsigned long*)((unsigned char*) start +
      ↪ i) == SEARCHED_VALUE)
18        unsigned long* field_ptr = (unsigned long*)
      ↪ ((unsigned char*) vm_start + i);
19  }
20 }
```

Listing 5: DHA using memory lookup

Even if this memory lookup fulfills the obfuscation goals, which are modifying a Java field value without using anything from the Java code, it still have two main drawbacks. First, the lookup has a high time overhead: to modify a single field, the native code has to scan the whole heap, which can grow to tens or hundreds of megabytes [4] (depending on the Android version). Second, the field has to be initialized to a unique value. This can lead to errors if the whole application code is not obfuscated at the same time. For example, an application can be obfuscated after adding a library that has been already obfuscated by its owner. In this case, fields from both the application and the library have been initialized to magic values that may be equal because when obfuscating the application, the potential library magic values are not known.

4.1.3. Advanced implementation: reflection. Native code can avoid the need for scanning the heap memory of a field by introspecting the obfuscated object itself. This requires an understanding of how the runtime stores objects and fields in memory and what native code has access to. This layout is presented in Figure 5. Native code has access to Java objects and fields through handles, respectively `jobject` and `jfieldID`. These are returned by JNI and no guarantees are given about their implementation.

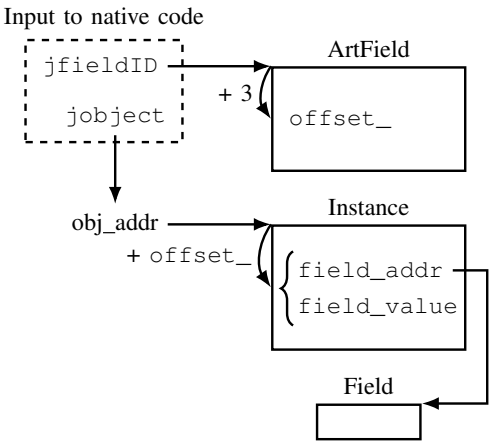

Figure 5: Memory layout of Java objects in Android

```
1 #define OFFSET_OF_OFFSET_FIELD_IN_ARTFIELD_CLASS 3
2 extern "C" JNIEXPORT jlong JNICALL retrieve_offset(
      ↪ JNIEnv *env, jobject thisObj) {
3  jclass cls = env->GetObjectClass(thisObj);
4  jfieldID fid = env->GetFieldID(cls, "polynomial",
      ↪ "I");
5  unsigned long offset_ = *((unsigned long*)fid +
      ↪ OFFSET_OF_OFFSET_FIELD_IN_ARTFIELD_CLASS);
6  return offset_;
7 }
```

Listing 6: Retrieving field offset

```
1 #define OFFSET 0x10
2 extern "C" JNIEXPORT void JNICALL reflection(JNIEnv
      ↪ *env, jobject thisObj) {
3  unsigned long* obj_addr = *(unsigned long**)
      ↪ thisObj;
4  unsigned long* field_value = &obj_addr[OFFSET/4];
5  *field_value = 0xeb31d82e;
6 }
```

Listing 7: DHA using reflection

However, by looking at the source code of the Android runtime, we observe that they are pointers. A `jfieldID` is a pointer to an instance of the `ArtField` class. This class is used, inside the runtime, to store information about the field such as its declaring class, its access flags (`private`, `public`) or even the offset of the field within an instance object (`offset_`). These pieces of information are set up by the class linker. A `jobject` is a pointer to an another pointer that refers to the effective object[5]. This object stores the addresses of the different fields of the object. For primitive fields, the field value is directly stored inside the object (instead of its address). Fields are sorted alphabetically, grouped by type. The order is wanted "relatively stable [...] so that adding new fields minimizes disruption of C++ version such as Class and Method."[6].

Thus, in order to implement a DHA trough reflection-like mechanism, the code has to first, retrieve the value `offset_`, which is the offset of the field address inside an object instance. This operation is realized in Listing 6. The `ArtField` instance of the field is retrieved at Line 4 and Line 5 retrieves the `offset_` value by accessing the third `long` of the `ArtField` instance. This offset (3) has been hardcoded at Line 1. Second, using the obtained `offset_`, the code modifies directly the field value. This is realized in Listing 7. The instance address of the object is retrieved on Line 3 by dereferencing the calling object. Then, at Line 4, the field address is obtained using the `offset_` value previously retrieved. It has to be noted that the first operation, Listing 6, requires to use JNI. To avoid being detected by JNI hooks, the value is computed and hardcoded in the Listing 7, at Line 1. Finally, the field value is set to `0xeb31d82e` at Line 5. Both listings have been successfully tested from Android 7.0 up to Android 10 without changing neither the value of `offset_` (0x10) nor the offset of `offset_` in `ArtField` class (3).

5. This allows the garbage collector to move object around the memory without having to change all references to it but only one.

6. AOSP source code, `class_linker.cc` file.

4.2. DHA detection

In order to detect Direct Heap Access, the analysis tools have to watch every read or write made to the heap during at runtime. This is done by disallowing, using `mprotect`, any access to the heap addresses when running native code. Then, when native code tries to access the heap, it generates a `SEGV` signal which can be retrieved. By parsing the internal structures of the garbage collector, the tool retrieves the type of the accessed value. Finally, the access is authorized and the execution is resumed. Section 5.2 presents the results obtained with this detection technique.

5. Experiment

In this section, the detection techniques proposed in Section 3 and 4 are used against real world applications in order to assess their effectiveness and usability and to find usages of the newly presented obfuscation techniques: BFO and DHA.

BFO, which targets firmware applications, is evaluated using a specially crafted dataset. This dataset is composed of a set of 16 firmwares, which contains 3479 precompiled applications (OATs) for Android Nougat (version 7.0 and 7.1.1)[7]. The complete list of firmwares is given in Appendix A.

DHA, which can be used on any application, is evaluated using two datasets: a subset of 100,018 applications from Androzoo [5] (which comprises around 9 million APKs collected since 2016 from more than 17 sources including the Google Play market), and the Android Malware Dataset (AMD) [6] (which comprises 24,552 labeled and classified malicious APKs among families of malware, dated from 2010 to 2016). Since AMD is composed only of malicious applications, comparing results for AMD and Androzoo allows to reveal characteristics specific to malware.

7. Firmwares have been downloaded from https://androidmtk.com/

5.1. BFO Detection

5.1.1. Search for removed bytecode. In order to detect if fully removed bytecode is already used in the wild (see Section 3.1), we have searched for a method containing quick code while not containing bytecode inside the pre-compiled applications of the firwmare dataset. This would have been the evidence of BFO usage. However, no such method has been found. This shows that BFO based on removing the bytecode is, at least for the firmwares we studied, not actively used in the wild.

We have also implemented a naive technique to detect partially removed bytecode. It consists in, first, computing, for each method, the ratio of the length of the bytecode over the length of the quick code and, then, checking if it exceeds a given threshold. Indeed, one could say that the number of assembly instructions used to represent a bytecode is bounded. However, compilers optimizations defeat this relation between bytecode and assembly. For example, compilers can decide that a method should be inlined when compiled or that a condition can be removed because it is always true or false. Thus, this method generates too much false positives to be usable.

5.1.2. Search for nopped bytecode. In order to evaluate the detection technique proposed in Section 3.1, we have implemented the following heuristic: for each method of the tested application, we compute the entropy of the bytecode. A small entropy reveals a nopped bytecode. To determine the threshold that reveals a nopped method, we have computed the entropy of the methods of all applications from a vanilla Android 7.0 (Nougat). This corresponds to 255,309 methods. These applications are not obfuscated, so their entropy should be higher than the threshold. The obtained entropy for each bytecode size is shown in Figure 6. For methods whose bytecode size is lower that 20, the entropy does not reveals anything and is too fluctuating to be able to set a threshold. Three thresholds are drawn on Figure 6: 0.1, 0.2 and 0.3. Results show that 0.1 is too strict while 0.3 generates too many false positives. Using a threshold of 0.2, only one method is falsely reported which is completely acceptable. Thus, by considering only bytecode of 20 bytes or more and by setting a threshold of 0.2, we should be able to detect nopped bytecode.

We applied this detection technique to the firmware dataset. As shown in Table 1, few methods have an entropy less than 0.2. We manually checked these methods by looking at their bytecode. We did not locate any usage of BFO. Listing 8 shows examples for three methods that are false positives. The code is not a nopped code since it is the initialization of several arrays. This initialization is composed of many times the same value, which lowers the entropy. However, this could have been a nopping pattern used to obfuscate applications. Thus, we believe that this method is able to detect nopped patterns to be confirmed by manual investigations.

5.1.3. Search for replaced bytecode. To evaluate if BFO consisting in replacing the whole bytecode by another correct one is already used in the wild, we have implemented the detection technique proposed in Section 3.2 and tested

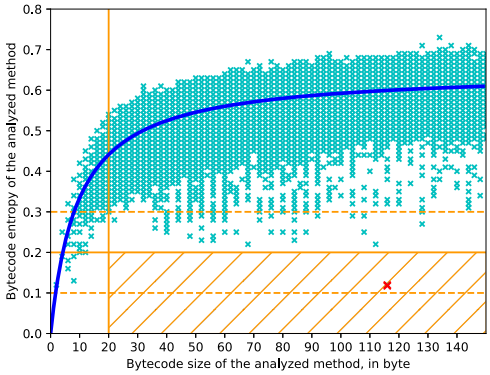

Figure 6: Bytecode entropy for methods of AOSP Android 7.0 APKs

TABLE 1: Nopped methods in firmware datatset

Firmwares		Total \ Entropy	< 0.1	< 0.2	< 0.3
Alcatel	APKs	338	0 0.00%	13 3.85%	138 40.83%
2 firmwares	Methods	2,716,821	0 0.00%	23 0.00%	614 0.02%
Archos	APKs	110	0 0.00%	2 1.82%	28 25.45%
1 firmwares	Methods	246,962	0 0.00%	2 0.00%	95 0.04%
Huawei	APKs	271	0 0.00%	9 3.32%	87 32.10%
3 firmwares	Methods	1,146,585	0 0.00%	9 0.00%	317 0.03%
Samsung	APKs	795	0 0.00%	6 0.75%	97 12.20%
5 firmwares	Methods	1,817,146	0 0.00%	12 0.00%	667 0.04%
Sony	APKs	1,412	0 0.00%	23 1.63%	341 24.15%
4 firmwares	Methods	5,463,229	0 0.00%	31 0.00%	1,547 0.03%
Wiko	APKs	188	0 0.00%	12 6.38%	81 43.09%
1 firmwares	Methods	1,709,624	0 0.00%	22 0.00%	365 0.02%
Total	APKs	3,114	0 0.00%	65 2.09%	772 24.79%
16 firmwares	Methods	13,100,367	0 0.00%	99 0.00%	3,605 0.03%

TABLE 2: Difference percentage for one firmware (21,521 methods, 43 applications)

Consecutive differences: 1	Threshold	0.00	0.25	0.50	0.90
	Number of detected methods	12715 (86.9%)	3606 (24.6%)	495 (3.4%)	445 (3.0%)
	Number of applications	19 (44.2%)	15 (34.9%)	10 (23.3%)	8 (18.6%)
Consecutive differences: 2	Threshold	0.00	0.05	0.20	0.30
	Number of detected methods	4347 (29.7%)	501 (3.4%)	53 (0.4%)	27 (0.2%)
	Number of applications	10 (23.3%)	8 (18.6%)	5 (11.6%)	4 (9.3%)
Consecutive differences: 5	Threshold	0.00	0.01	0.05	0.10
	Number of detected methods	748 (5.1%)	441 (3.0%)	55 (0.4%)	20 (0.1%)
	Number of applications	9 (20.9%)	8 (18.6%)	5 (11.6%)	3 (7.0%)

```
1 // Entropy: 0.197
2 public static final float[] horizontalFlipMatrix() {
3   return new float[] { -1.0F, 0.0F, 0.0F, 0.0F, 0.0F
      ↪ , 1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F,
      ↪ 1.0F, 0.0F, 0.0F, 1.0F };
4 }
5 // Entropy: 0.180
6 public static final float[] identityMatrix() {
7   return new float[] { 1.0F, 0.0F, 0.0F, 0.0F, 0.0F,
      ↪  1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F,
      ↪ 0.0F, 0.0F, 0.0F, 1.0F };
8 }
9 // Entropy: 0.197
10 public static final float[] verticalFlipMatrix() {
11   return new float[] { 1.0F, 0.0F, 0.0F, 0.0F, 0.0F,
      ↪  -1.0F, 0.0F, 0.0F, 0.0F, 0.0F, 1.0F, 0.0F,
      ↪ 0.0F, 1.0F, 0.0F, 1.0F };
12 }
```

Listing 8: Example of false positive for nopped bytecode search

```
1 push {r3,r4,r5,r6,r7}          push {r3,r4,r5,r6,r7}
2 movs r4, r5                    movs r4, r5
3 movs r2, r5                    movs r2, r5
4 add r2, sp, #0x3c0           | pop r4,r5,r6,r7
5 bhs #0xffffff54                bhs #0xffffff54
6 ldrb r0, [r7, #3]              ldrb r0, [r7, #3]
7 pop {r6, r7}                   pop {r6, r7}
8 lsrs r1, r6, #0xb              lsrs r1, r6, #0xb
9 stm r0!, {r0,r1,r2,r3}         stm r0!, {r0,r1,r2,r3}
10 adr r4, #0x3c8              <
11 hint #8                       hint #8
12 ldrh r3, [r6, r7]             ldrh r3, [r6, r7]
13                             > ldr r0, [pc, #0x23c]
14 adds r2, #0xf6              | ldrh r2, [r6, #0x16]
15 stm r0!, {r0,r1,r2,r4,r6}     stm r0!, {r0,r1,r2,r4}
16 lsls r2, r6, #0xb             lsls r2, r6, #0xb
17 ldrb r7, [r0, #0x1c]          ldrb r7, [r0, #0x1c]
18 lsls r4, r0, #0x1d            lsls r4, r0, #0x1d
```

Listing 9: Example of differences

it over the firmware dataset. For each precompiled application (OAT file), we extracted the bytecode file (DEX) from the OAT file. Then, we recompiled it using the emulator provided by Google. We carefully chose the emulator to reflect the Android version and the processor architecture used by the real smartphone (ARM emulator for Nougat version 7.0). Compiling using the same environment as the firmware constructor is impossible since applications are cross-compiled on vendor computers and no documentation is available on their build systems.

Finally, we compared the obtained assembly code with the one of the firmware. If no BFO techniques has been used, they should be equal. However, in practice, a compiler is very influenced by the configuration of a particular system and many of them use non-deterministic algorithms [7]. Thus, the obtained assembly code and the firmware's one are slightly different, for almost all methods.

In order to investigate how much the codes are different, we first tried to use state-of-the-art binary diffing tools [7], [8], such as bindiff[8] or diaphora[9]. However, they did not achieve to detect more accurately whether codes are the same. Indeed, these tools rely heavily on the call-graph of the analyzed codes which is almost nonexistent for quick code: due to its object and framework oriented compilation, all calls are indirect and cannot be resolved statically.

Thus, we have built a custom comparison technique. The code is disassembled and operands, which usually correspond to offsets (immediate values) or registers that are very likely to change between two compilations, are removed. This results in an abstract version of the assembly code. Then, we proceeded to a classical diff where each assembly instruction constitutes a line. Finally, we computed the following ratio: number of differences over the number of lines. If, for a method, this ratio is over a threshold, the method is considered obfuscated.

This ratio has been calculated for pre-compiled applications of one firmware. The results obtained with this ratio are shown in the first three rows of Table 2 ("Consecutive differences: 1"). For 24.6 percent of the methods, more that one instruction out of four differ. By manually investigating the differences, we saw that they are due to compilation, that is they are false positive. When using high threshold (colored in red in Table 2) the number of detected methods is reduced. In the same time, this increases the number of false negatives and thus is not suitable.

However, by looking at the generated diffs, such as Listing 9, we saw that most of the differences are composed of the addition of one instruction, or the replacement of an equivalent. Thus, in order to improve the ratio, we count differences only when several are consecutive. Table 2 shows results when differences are taken into account when 2 or 5 are consecutive (colored in blue in Table 2). This reduces the number of detected methods which allowed us to manually check all of them. Finally, this manual investigation showed that none of them were true positives.

5.1.4. Future work on searching for BFO usage. Thus far, no BFO usage has been found in the wild for BFO consisting in removing or nopping the bytecode. For the replacement case, no suitable detection technique is known. Thus, more specific techniques need to be developped. State-of-the-art binary diffing tries to match assembly functions between two code bases. When trying to detect BFO usage, we need to verify that a specific bytecode method corresponds to a specific assembly method. The potential association is already known.

5.2. Statistics on DHA usage

To assess how much DHA is used in the wild, maliciously or not, we used the detection technique described in Section 4 on two datasets: Androzoo and AMD. Indeed, we need to execute each application in order to detect DHA usage. It would be very difficult to setup on firmware dataset, as it requires a different smartphone model for each firmware. The detection has been implemented for ARMv8 and Android version 7.0. The datasets were first filtered to keep only the compatible APKs, and we checked that these applications can be launched correctly. Column "ARMv8" of Table 3 reports the number of applications obtained after applying this filter.

We analyzed these filtered datasets and logged all DHA, *i.e.*, each time the heap was accessed from the native code. Note that each application was run from

8. https://www.zynamics.com/bindiff/manual/
9. https://github.com/joxeankoret/diaphora/

TABLE 3: Number of DHAs detected

Dataset	Total	ARMv8	DHA	DHA without system libs
Androzoo [5]	100,018	10,661	8,158 (76.5 %)	4,021 (37.7 %)
AMD [6]	24,552	349	194 (55.6 %)	103 (29.5 %)
Total	124,570	11,010	8,352 (75.9 %)	4,124 (37.5%)

TABLE 4: Classes and libraries detected to be using DHA

Dataset	System libraries		WebView		Other	
	samples	classes	samples	classes	samples	classes
Androzoo [5]	74.7%	1,797	37.3%	1,424	0.4%	7
AMD [6]	54.7%	154	29.5%	221	0%	0

only the main activity and without any user interaction. Consequently, the results presented in Table 3 are a lower bound. For each DHA, we logged the class of the accessed value and the name of the library performing the access, from `/proc/self/maps`.

Globally, between 55% and 76% of the applications performed DHAs. This lower bound shows that DHA cannot be ignored when building an analysis tool. When investigating which libraries have performed DHAs, we noticed that most accesses are done by systems libraries (*e.g.* `libc.so`, `boot.oat`, `libandroid_runtime.so`). However, we have still detected that 37% of applications perform DHAs using custom libraries.

A comparison of the statistics retrieved for Androzoo and AMD datasets showed that DHA usage does not discriminate a malicious behavior from a benign one. In fact, according to the name of the libraries performing DHA, it seems to be used mostly to increase performance.

We investigated the name of the classes accessed by DHA, the number of unique class names is reported in Table 4. As expected, system libraries access a large variety of object of different classes as these libraries are part of the runtime internals. Additionally, we separated a specific library, `WebView`, because it manipulates a lot of internal objects of the browser. Finally, remaining libraries modifies seven different classes. Almost every sample uses `[F`, `String`, `[B` or `ByteArrayInputStream` which confirms that developers mainly use DHA as Google recommends without bypassing their guidelines [9]. We notably notice that one library, `conscrypt`[10], accesses the `OpenSSLX509Certificate` and `OpenSSLX509CertificateFactory` classes using DHA.

This is comforting in that DHA is not yet used as a way to bypass analysis, even in the security community [10]. However, due to the high number of benign DHA, few malicious ones could be hidden and remain undetected. This highlights the need for tools and methods that take into account this kind of accesses.

6. Related work

There are mainly two types of obfuscation that use assembly code to obfuscate Android applications: packing and ahead-of-time compilation (AOTC)-based bytecode hiding.

10. https://github.com/google/conscrypt

Packing consists in storing the Dalvik bytecode ciphered before dynamically deciphering and loading it at runtime. This mechanism relies on native code that directly modifies Android's internal structures to deploy new bytecode. These modifications can occur at any time of the bytecode loading process or before the execution. Packing has been extensively studied [11]–[15], and packing DEX bytecode using native code is a popular technique: Duan *et al.* [16] reported that an average of 13.89% of malware in the wild between 2010 and 2015 used packing techniques to hide malicious behavior.

The ahead-of-time compilation (AOTC)-based bytecode hiding scheme is a recently described obfuscation technique [10] that aims at hiding the bytecode of sensitive methods from both static and dynamic analyses. It is composed of three main steps performed before releasing the APK file. First, the bytecode of obfuscated methods is removed from the DEX file. Then, the bytecode of these methods is compiled using a custom compiler, producing native code. Finally, calls to obfuscated methods are transformed into JNI calls. Unlike the packing method, the bytecode is never deciphered and, thus, is never directly available to analysis tools. The code is only present in its compiled form.

Thus, as far as we know, no article has ever studied obfuscation made specifically for pre-compiled applications. Indeed, AOTC-based bytecode hiding uses the compilation mechanism to hide some specific methods while BFO (see Section 3) uses it to remove potentially all the bytecode. Also, it takes care of not being detected in order to keep the real application purpose hidden. Moreover, works about packing study the interface between bytecode and assembly code at the execution level: how assembly can interfere with the bytecode. To go further, this article proposes to study how assembly code can interfere with the data used by the bytecode (see DHA in Section 4).

7. Conclusion

This article presents two new obfuscation techniques based on the use of native code and associated detection techniques. Our experiments show the feasibility and the stealthiness of hiding code inside assembly compiled from bytecode. The study of applications in the wild also shows that bypassing the JNI interface using DHA techniques is already used by applications but not yet for malicious intents. Experiments about BFO usage shows that no simple form of BFO is used in the wild. Even if no malicious uses of these new obfuscation techniques have been found, their feasibility shows that analysis tools should now pay attention to them in case they start being used by malware.

References

[1] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *ACM SIGSAC Conference on Computer and Communications Security*, no. 18. Toronto, Canada: ACM, oct 2018, pp. 1137–1150.

[2] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *USENIX Security Symposium*, no. 26. Vancouver, Canada: USENIX, aug 2017, pp. 289–306.

[3] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, no. 44. Atlanta, USA: IEEE, sep 2014, pp. 180–191.

[4] "Android compatibility definition document," https://source.android.com/compatibility/cdd.html, 2020.

[5] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *International Conference on Mining Software Repositories*, no. 13. Austin, Texas: ACM, may 2016, pp. 468–471.

[6] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware1," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, no. 14. Bonn, Germany: Springer, jul 2017, pp. 252–276.

[7] T. Dullien, E. Carrera, S.-M. Eppler, and S. Porst, "Automated attacker correlation for malicious code," Tech. Rep., mar 2010.

[8] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment*, no. 1. Dortmund, Germany: Gesellschaft für Informatik, jul 2004, pp. 161–173.

[9] *JNI tips*, Android, 2019, https://developer.android.com/training/articles/perf-jni#primitive-arrays.

[10] J. Bao, Y. He, and W. Wen, "Droidpro: An aotc-based bytecode-hiding scheme for packing the android applications," in *IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ IEEE International Conference On Big Data Science And Engineering*. New York, USA: IEEE, aug 2018, pp. 624–632.

[11] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: toward extracting hidden code from packed android applications," in *European Symposium on Research in Computer Security*, no. 20. Vienna, Austria: Springer, nov 2015, pp. 293–311.

[12] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in *International Symposium on Recent Advances in Intrusion Detection*, no. 18. Kyoto, Japan: Springer, dec 2015, pp. 359–381.

[13] Y. Liao, J. Li, B. Li, G. Zhu, Y. Yin, and R. Cai, "Automated detection and classification for packed android applications," in *International Conference on Mobile Services*. San Francisco, USA: IEEE, jun 2016, pp. 200–203.

[14] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *International Conference on Software Engineering*, no. 39. Buenos Aires, Argentina: IEEE, may 2017, pp. 358–369.

[15] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with tiro," in *USENIX Security Symposium*, no. 27. Baltimore, USA: USENIX, aug 2018, pp. 1247–1262.

[16] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation," in *Network and Distributed System Security Symposium*, no. 25, San Diego, USA, feb 2018.

[17] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, G. McGraw, Ed. Addison-Wesley Professional, jul 2009, no. 1.

[18] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

Appendix A.
List of tested firmwares

TABLE 5: List of tested firmwares

Brand	Phone model	Android version
Alcatel	1T 10	7.0
	OneTouch A3 Plus 5011A	7.0
Archos	50f Neon	7.0
Huawei	Ascend Mate 9 MHA-AL00	7.0
	Enjoy 7 Plus TRT-TL10A	7.0
	P10 VRT-AL00	7.0
Samsung	Galaxy A3 SM-A310M	7.0
	Galaxy C7 Pro SM-C710F	7.1.1
	Galaxy Note 5 SM-N920A	7.0
	Galaxy S6 Edge SM-G925S	7.0
	Galaxy A5 SM-A510M	7.1.1
Sony Xperia	Touch G1109	7.0
	L1 Dual G3312	7.1.1
	M2 Aqua D2403	7.0
	Z5 Premium E6853	7.0
	Z5 501SO	7.1.1
Wiko	Jerry 2	7.0