Computers, Materials & Continua
DOI: 10.32604/cmc.2022.026395
Article

Tech Science Press

Deobfuscating Mobile Malware for Identifying Concealed Behaviors

Dongho Lee, Geochang Jeon, Sunjun Lee and Haehyun Cho*

Soongsil University, Seoul, 06978, Korea
*Corresponding Author: Haehyun Cho. Email: haehyun@ssu.ac.kr
Received: 24 December 2021; Accepted: 16 March 2022

Abstract: The smart phone market is continuously increasing and there are more than 6 billion of smart phone users worldwide with the aid of the 5G technology. Among them Android occupies 87% of the market share. Naturally, the widespread Android smartphones has drawn the attention of the attackers who implement and spread malware. Consequently, currently the number of malware targeting Android mobile phones is ever increasing. Therefore, it is a critical task to find and detect malicious behaviors of malware in a timely manner. However, unfortunately, attackers use a variety of obfuscation techniques for malware to evade or delay detection. When an obfuscation technique such as the class encryption is applied to a malicious application, we cannot obtain any information through a static analysis regarding its malicious behaviors. Hence, we need to rely on the manual, dynamic analysis to find concealed malicious behaviors from obfuscated malware. To avoid malware spreading out in larger scale, we need an automated deobfuscation approach that accurately deobfuscates obfuscated malware so that we can reveal hidden malicious behaviors. In this study, we introduce widely-used obfuscation techniques and propose an effective deobfuscation method, named ARBDroid, for automatically deobfuscating the string encryption, class encryption, and API hiding techniques. Our evaluation results clearly demonstrate that our approach can deobfuscate obfuscated applications based on dynamic analysis results.

Keywords: Android; obfuscation; deobfuscation; android reversing

1 Introduction

The use of mobile devices is continually increasing, and more than 6 billion mobile phone users worldwide will overtake landline use with the 5G technology [1,2]. Among them, Android is the most popular mobile platform, occupying 87% of the market share [3]. Naturally, a ton of emerging malware is threating numerous Android users to leak personal information and steal sensitive information [4]. Therefore, it is critical to detect and analyze malicious applications for identifying malicious behaviors [5–8]. To this end, approaches to detect malicious applications in the market or in the wild have drawn attention the research community. However, recently attackers make the analysis difficult (or impossible) by employing obfuscation techniques to delay or evade detection [9,10].

When an obfuscation technique such as the class encryption is applied to a malicious application, we cannot obtain any information through a static analysis regarding its malicious behaviors. Hence, we need to rely on the manual, dynamic analysis to find concealed malicious behaviors from obfuscated malware. However, analyzing an obfuscated malicious application is a time-consuming and error-prone task, and thus, in the community it is great importance of to accurately deobfuscate the obfuscated malware to avoid malware spreading out in larger scale.

In this paper, we propose an automated deobfuscator, named ARBDroid, that can deobfuscate the string encryption, class encryption, and API hiding obfuscation techniques by executing and dynamically analyzing the obfuscated malicious applications. In this work, we specifically focus on the three obfuscation techniques because these techniques make static analysis infeasible by hiding important information for the reversing applications with various cryptographic algorithms.

ARBDroid implemented on the Android runtime directly executes obfuscated applications and extracts instructions of Android applications including other information such as string values and calling APIs. Such dynamically generated logs are going to be used to deobfuscate malware. After extracting the code and data, ARBDroid modifies the original *classes.dex* file (i.e., the main executable file of Android applications) for inserting decrypted strings, revealed API calls, and decrypted files of classes.

To evaluate the effectiveness of ARBDRoid, we collected 10,000 in-the-wild malware appeared from 2018 to 2020. By using the real-world dataset, we show that ARBDroid can effectively deobfuscate the three obfuscation techniques. Also, by using ARBDroid, we found that 1,628 malicious applications out of 10,000 were hiding sensitive API calls for obtaining information such as the device status, MAC address of a device, and installed packages in a device.

In summary, the contributions of this paper are as follows.

- We propose ARBDroid that automatically deobfuscates obfuscated Android malicious applications by dynamically analyzing them. ARBDroid can deobfuscate the string encryption, class encryption, and API hiding obfuscation techniques.
- We analyze obfuscated real-world malware to demonstrate the effectiveness of ARBDroid. Our evaluation results clearly show that ARBDroid can be used to identify hidden malicious behaviors by automatically deobfuscating malware.

2 Background

In this section, we introduce widely-used obfuscation techniques and an example of obfuscated malicious application code.

2.1 Obfuscation Techniques

Obfuscation techniques are used to make it challenging for the analyst to analyze the code. Because Android applications are easy to decompile (and thus can be easily analyzed), obfuscation is essentially necessary to protect applications from security threats such as the repackaging attack [11]. Most of the commercial obfuscation tools support string encryption, class encryption, and API hiding techniques [12], and thus, the attackers can easily use them by using commercial obfuscators. In addition, the three obfuscation techniques are the key technique that can be used to conceal malicious behaviors from the security analysts by hiding important information from any static analysis technique. Therefore, in this work, we focus on the three obfuscation techniques.

2.1.1 String Encryption

String encryption is used to hide strings in an application by encrypting strings. Such encrypted strings are decrypted when they are used during the run time [13]. Because strings typically contain important information such as the secret key, the string encryption is one of the mostly used obfuscation technique. Also, since strings are encrypted, the string encryption can make the static analysis very difficult.

2.1.2 Class Encryption

Class encryption is a technique of encrypting a class to protect it from being analyzed [14]. The encrypted class must be decrypted by an application before loaded. Because this technique encrypts an entire class, it can effectively work against the static analysis.

2.1.3 API Hiding and Hide Access

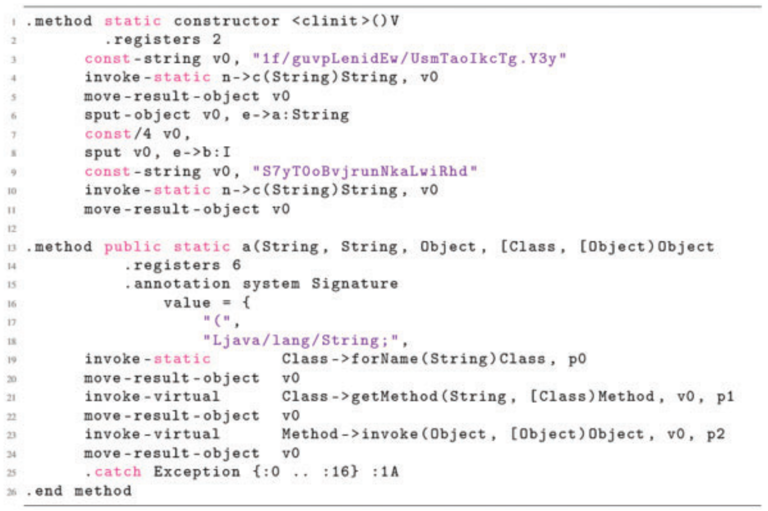
The API hiding is a technique to hide API calls, which in general is implemented by using the Java Reflection and Java Native Interface (JNI) [15]. Hide access is also used to protect API calls and other accesses to members or methods of any classes [16]. If an application is protected by the API hiding or hide access technique, attackers cannot statically identify behaviors of the application.

2.1.4 Identifier Renaming

Renaming is transforming all class, method, and field names into random strings [17]. Because developers tend to give meaningful identifiers to such variables in general, the renaming technique aims to make an application difficult to analyze by obfuscating variable names.

2.2 An Example of Obfuscated Malware

If security analysts can identify strings or API calls in malware, malicious behaviors of the malware can be easily revealed. Therefore, attackers usually employ obfuscation techniques to prevent or delay detection. Fig. 1 shows an obfuscated, malicious Android application's code in a form of the smali code.



```
.method static constructor <clinit>()V
        .registers 2
      const-string v0, "1f/guvpLenidEw/UsmTaoIkcTg.Y3y"
      invoke-static n->c(String)String, v0
      move-result-object v0
      sput-object v0, e->a:String
      const/4 v0,
      sput v0, e->b:I
      const-string v0, "S7yTOoBvjrunNkaLwiRhd"
      invoke-static n->c(String)String, v0
      move-result-object v0

.method public static a(String, String, Object, [Class, [Object)Object
          .registers 6
          .annotation system Signature
              value = {
                  "(",
                  "Ljava/lang/String;",
      invoke-static        Class->forName(String)Class, p0
      move-result-object   v0
      invoke-virtual       Class->getMethod(String, [Class)Method, v0, p1
      move-result-object   v0
      invoke-virtual       Method->invoke(Object, [Object)Object, v0, p2
      move-result-object   v0
      .catch Exception {:0 .. :16} :1A
.end method
```

Figure 1: An example of smali code of obfuscated Android malware

The smali code is a disassembled implementation of the dex format used by Android runtime (Dalvik virtual machine) [18]. In the code snippet, we cannot find a plain string and which APIs are called because the string encryption and API hiding techniques are used in the malware. Consequently, we cannot find any specific behavior of the malware from the code. As such, obfuscation techniques are very effective to prevent detection of malicious behaviors, it is critical to deobfuscate obfuscated malicious applications to detect them in a timely manner.

3 Overview

Since commercial obfuscation tools such as Dexguard [19], VMprotect [20], Themida [21], and Dexprotector [22] are available in the market, attackers can easily obfuscate malware by using them. However, there has not been a complete study on the automated deobfuscation against such commercial obfuscation tools [23]. In this work, we focus on deobfuscating string encryption, class encryption, API hiding techniques based on analysis results of commercial obfuscation tools. Specifically, the goal of this work is as follows: (G1) Deobfuscate the string encryption technique; (G2) Deobfuscate the class encryption technique; and (G3) Deobfuscate the API hiding technique. Because the renaming identifier technique does not critically affect the analysis process and cannot convert back to the original identifiers, we exclude deobfuscating the identifier renaming technique in this work.

In order to achieve the goal, we dynamically analyze malware, extracting important information to deobfuscate malware from the Android runtime. Extracted information includes decrypted strings, classes, and API information protected by the API hiding technique. Encrypted data and code (including target API information) must be revealed when an application is executing due to the nature of program execution. Based on the extracted information, we reconstruct smali code of obfuscated malware. This approach has the following advantages: (1) By directly analyzing obfuscated malware, there is no semantic gap between deobfuscated information and the malware; and (2) By executing malware on the Android runtime, we can bypass anti-analysis techniques such as the anti-emulation technqiue. In the following section, we introduce the detail design of our system to automatically deobufscate obfuscated Android malware.

4 Design

In this section, we describe the design of ARBDroid that can automatically deobfuscate the string encryption, class encryption and API hiding obfuscation techniques. Fig. 2 illustrates the architecture of ARBDroid. ARBDroid executes an obfuscated application, extracting information needed to deobfuscate the application such as API calls, parameters used to call methods and return values from methods. Also, ARBDroid extracts executing instructions (i.e., byte-code) of an application. The extracted information is stored in the component manger in Fig. 2.

Then, the instruction monitor translates the information to be byte-code so that the translated byte code can be inserted to the original *classes.dex* file. Lastly, ARBDroid generates a *Deobfuscated.dex* file that contains deobfuscated information (i.e., decrypted strings, decrypted classes, and revealed API calls) so that security analysts can perform analysis for identifying malicious behaviors. It is worth noting that the deobfuscated applications can be used in static analysis tools such as FlowDroid [24], which was not possible because of the obfuscation techniques used in malicious applications.

4.1 Deobfuscating the String Encryption

The string encryption technique encrypts all strings of an application. To restore the encrypted strings during the runtime, a decryption module is stored in the application as well. Therefore, in order to statically deobfuscate encrypted strings, we must find and analyze the decryption algorithm in an application. However, such approach takes a lot of time to analyze and error-prone. Also, each obfuscation tool uses different encryption algorithms, and thus, analysts must repeat the same process for every application. We overcome this problem by logging all string objects generated by an application. Then, the printed strings inserted to the classes.dex file of the application as the last step. Fig. 3 illustrates the deobfuscation process for encrypted strings with ARBDroid.

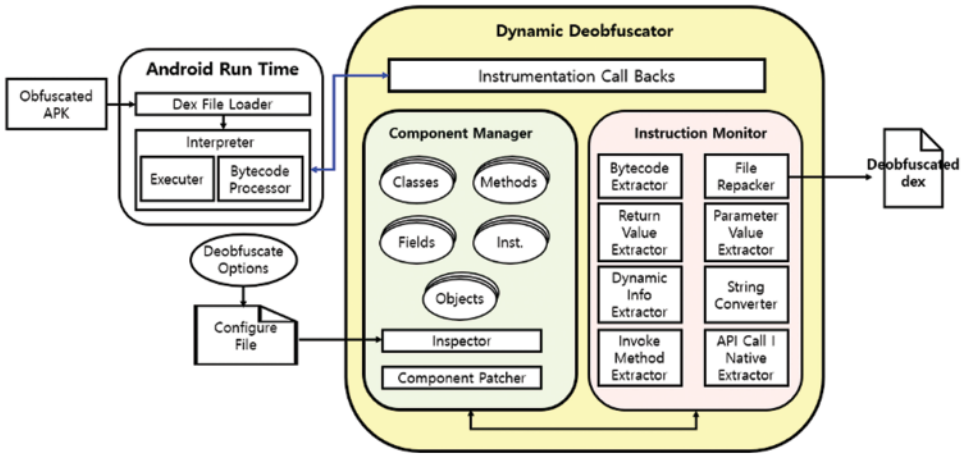


Figure 2: The architecture of the proposed deobfuscator

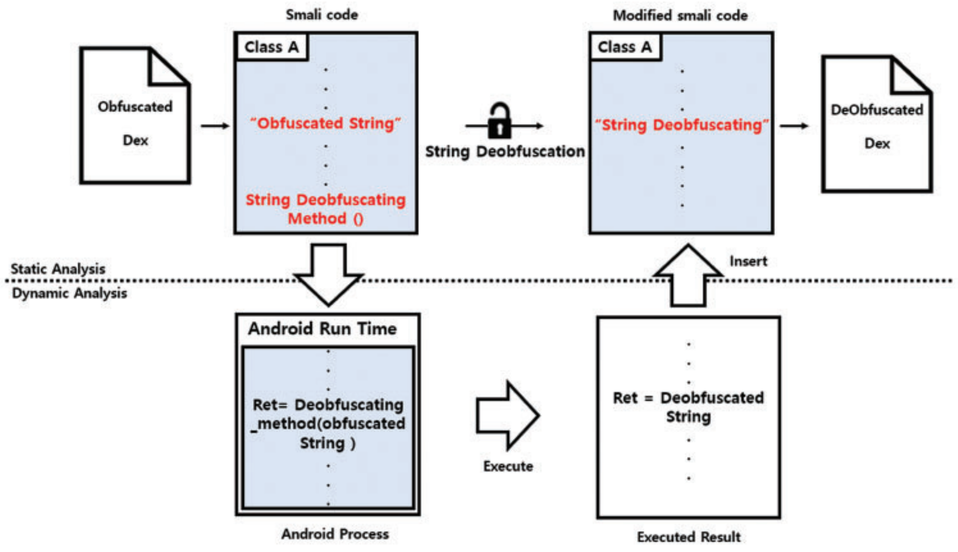


Figure 3: Our deobfuscation approach for the string encryption technique

To be specific, our deobfuscation method identifies all string-type return values returned from all method calls in an application. Next, it logs strings with other information such as instructions which call a method that returns string-type values. Lastly, ARBDroid finds a correct place where the decrypted string should be placed in the original classes.dex file and inserts the decrypted string to the place. Consequently, security analysists can see plain texts in all points where encrypted strings are used as a result of the deobfuscation process.

4.2 Deobfuscating the Class Encryption

Fig. 4 shows how a typical class encryption technique operates when it encrypts a class and decrypts the class during the runtime.

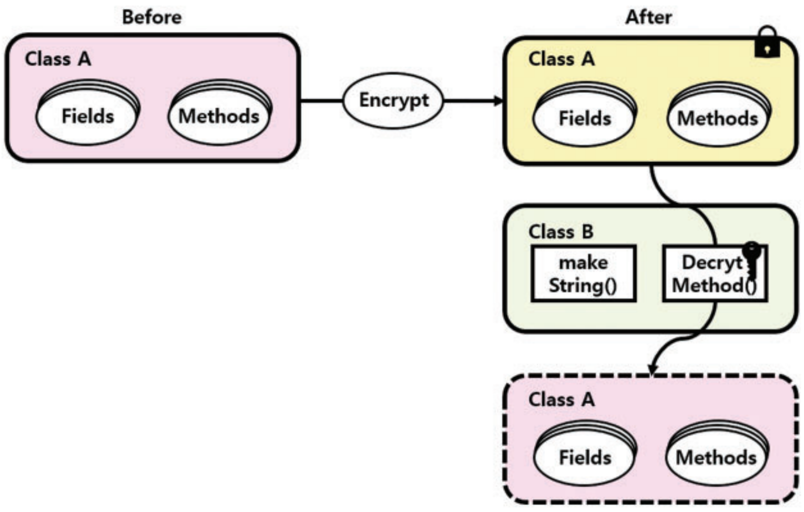


Figure 4: How the class encryption works in general

As we discussed in Section 3, the encrypted code must be restored when the class is loaded. Therefore, if we can interrupt the process when it loads the decryted class, we can obtain the original class. Fig. 5 explains the process of deobfuscating a class encryption. At the moment when the encrypted class is decrypted and loading to run, ARBDroid extracts the decrypted class as a .dex file. The deobfusction process consists of four steps: (1) It dynamically activates an obfuscated application and loads the application into the memory; (2) It extracts the decrypted classes that are going to be loaded into the memory; (3) It insert the decrypted classes into the deobfuscated.dex file.

4.3 Deobfuscating the API Hiding

Fig. 6 illustrates the deobfuscation process for the API hiding technique, which consists of seven steps. If the API hiding technique is applied onto an application, we cannot statically find which API is called in the application. Therefore, similar to the other deobufscation approaches discussed in the previous sections, we find which APIs are called through executing an application at the Android runtime. Every time when an API is called during the runtime of an application, ARBDroid extracts instructions of the application that calls APIs. To be specific, ARBDroid logs out an API name (including the class name) and parameters used to call the API. After identifying the hidden information, ARBDroid modifies the original dex file to insert the extracted information to a point where the API hiding technique is applied. Because ARBDroid prints out general instructions of an application while it is running, we can find an exact instruction that calls an APIs by matching logged instructions with instructions of the classes.dex file. In summary, the deobfuscation process for the API hiding is as follows: (1) Decompiling the obfuscated .dex file; (2) Executing the obfuscated application; (3) Extracting information regarding API calls including the class name, method name, and parameter values; (4) Rewriting the classes.dex file by using the identified information.

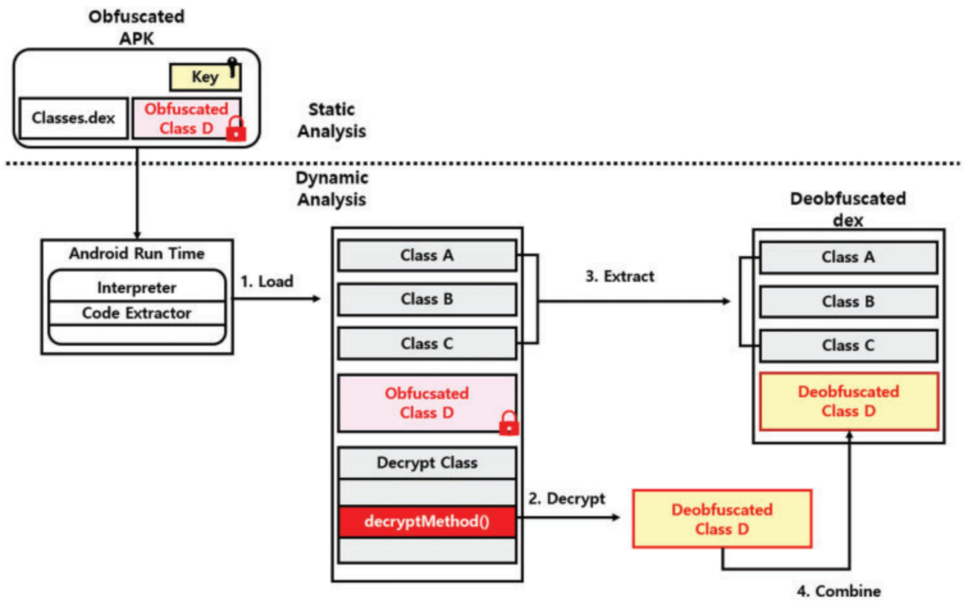


Figure 5: Our deobfuscation approach for the class encryption technique

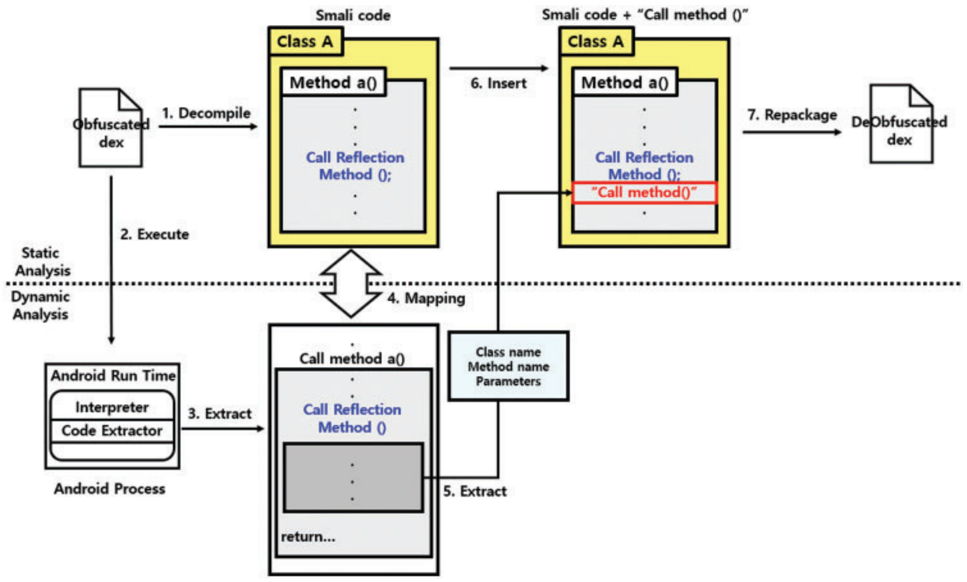


Figure 6: Our deobfuscation approach for the API hiding technique

5 Evaluation

In this section, we evaluate ARBDroid, answering the following research questions.

RQ1. *How does ARBDroid effectively deobfuscate each obfuscation technique targeted in this work?*

RQ2. *Can ARBDroid identify hidden sensitive API calls in real-world malware?*

5.1 Experiment Environment

We performed the evaluation on a Google Pixel 2 XL 64GB with Android version 8.1.0 and on a PC that has Intel i5-8500 3.00 GHz CPU with 24GB RAM. To evaluate ARBDroid, we implemented it on the Android runtime version 8.1.0 with C++, Java, and Python, which consists of 3,525 SLoC in total. Also, as Tab. 1 shows, we used 10,000 in-the-wild malware collected from VirusShare for our evaluation. Lastly, we used Android Monkey to automate various executions through randomly generated inputs.

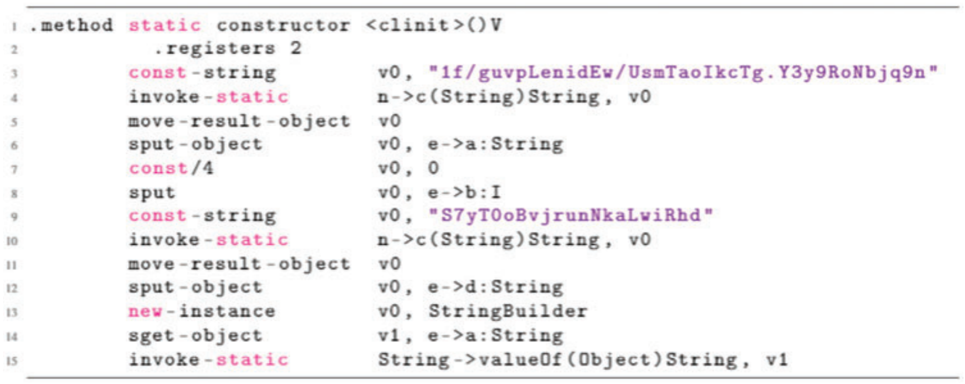
Table 1: Malware dataset used for our evaluation

Year	2018	2019	2020	Total
# of Malware	4,000	3,000	3,000	10,000

5.2 How Does ARBDroid Effectively Deobfuscate Each Obfuscation Technique Targeted in This Work?

5.2.1 Deobfuscating the String Encryption

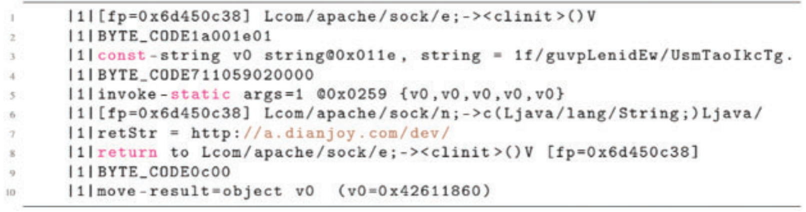
Fig. 7 shows smali code of a malicious application to which string encryption was applied. As in Line 3 and Line 9, strings are encrypted and thus, security analysts cannot recognize any information



```
1  .method static constructor <clinit>()V
2            .registers 2
3          const-string        v0, "1f/guvpLenidEw/UsmTaoIkcTg.Y3y9RoNbjq9n"
4          invoke-static       n->c(String)String, v0
5          move-result-object  v0
6          sput-object         v0, e->a:String
7          const/4             v0, 0
8          sput                v0, e->b:I
9          const-string        v0, "S7yTOoBvjrunNkaLwiRhd"
10         invoke-static       n->c(String)String, v0
11         move-result-object  v0
12         sput-object         v0, e->d:String
13         new-instance        v0, StringBuilder
14         sget-object         v1, e->a:String
15         invoke-static       String->valueOf(Object)String, v1
```

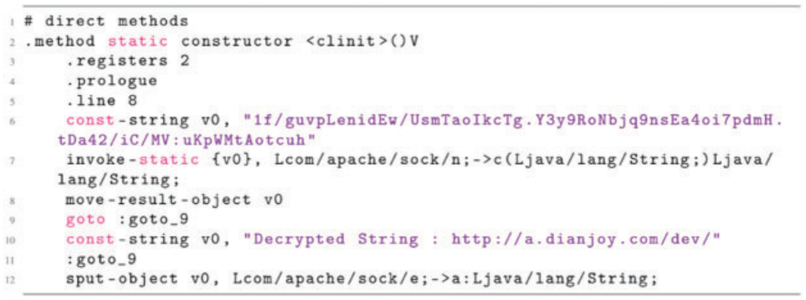
Figure 7: Smali code of a malicious application to which the string encryption is applied

Fig. 8 is a part of the log extracted from ARBDroid. In the log, when the return value of a method is java.lang.String type, ARBDroid prints the value with a string of *retStr* (Line 7). As in Line 10 of Fig. 9, after extracting the original string, ARBDroid inserts the original string to the original classes.dex file and stores it as *Deobfuscated.dex* file.



```
1      |1|[fp=0x6d450c38] Lcom/apache/sock/e;-><clinit>()V
2      |1|BYTE_CODE1a001e01
3      |1|const-string v0 string@0x011e, string = 1f/guvpLenidEw/UsmTaoIkcTg.
4      |1|BYTE_CODE711059020000
5      |1|invoke-static args=1 @0x0259 {v0,v0,v0,v0,v0}
6      |1|[fp=0x6d450c38] Lcom/apache/sock/n;->c(Ljava/lang/String;)Ljava/
7      |1|retStr = http://a.dianjoy.com/dev/
8      |1|return to Lcom/apache/sock/e;-><clinit>()V [fp=0x6d450c38]
9      |1|BYTE_CODE0c00
10     |1|move-result=object v0  (v0=0x42611860)
```

Figure 8: Log of ARBDroid when string-type value is returned from a method



```
# direct methods
.method static constructor <clinit>()V
    .registers 2
    .prologue
    .line 8
    const-string v0, "1f/guvpLenidEw/UsmTaoIkcTg.Y3y9RoNbjq9nsEa4oi7pdmH.
   tDa42/iC/MV:uKpWMtAotcuh"
    invoke-static {v0}, Lcom/apache/sock/n;->c(Ljava/lang/String;)Ljava/
   lang/String;
    move-result-object v0
    goto :goto_9
    const-string v0, "Decrypted String : http://a.dianjoy.com/dev/"
    :goto_9
    sput-object v0, Lcom/apache/sock/e;->a:Ljava/lang/String;
```

Figure 9: Smali code of Deobfuscated .dex file that contains decrypted strings

5.2.2 *Deobfuscating the Class Encryption*

Fig. 10 shows smali code of malware of which classes were encrypted. When we disassembled the application, we could find only 3 class files: qihoo.smali, util.smali, and qihoo360.smali.

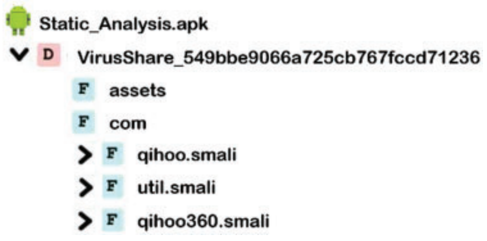


Figure 10: Malware with Class Encryption technique applied

Fig. 11 shows the *Deobfuscated.dex* file generated by ARBDroid, which contains four more classes which were decrypted and loaded during the run time. ARBDroid extracted each class file when they were loaded into the memory.

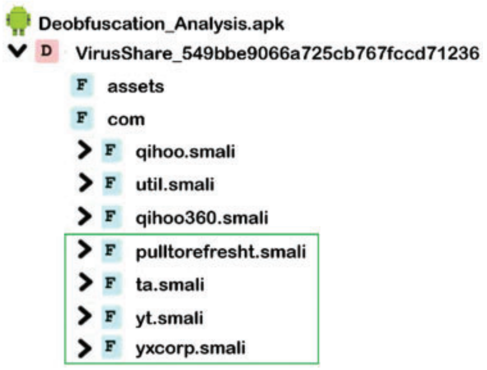


Figure 11: Class files extracted by ARBDroid

5.2.3 *Deobfuscating the API Hiding*

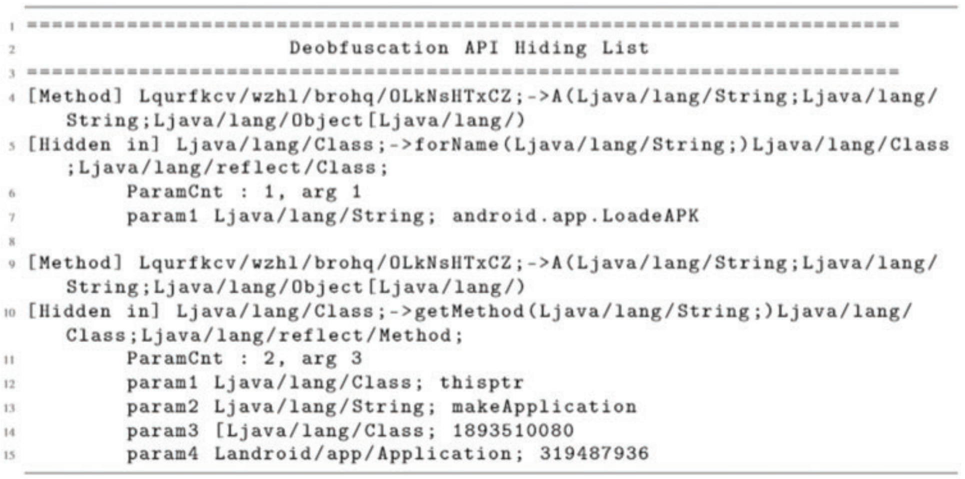
Fig. 12 shows smali code of malware to which the API hiding technique was applied. As in the snippet, the API call is hidden to prevent the static analysis. Fig. 13 is a part of the log extracted by ARBDroid. The log includes the hidden method call structure: the target method name, class name, parameters. Fig. 14 shows the result of deobfuscating hidden API calls including the one in Fig. 12. By using this information, ARBDroid modifies the classes dexfile to generate the *Deobfuscated.dex file* to provide the concealed API call information to analysts.

```
.method public static a(String, String, Object, [Class, [Object)Object
          .registers 6
    invoke-static        Class->forName(String)Class, p0
    move-result-object   v0
    invoke-virtual       Class->getMethod(String, [Class)Method, v0, p1, p3
    move-result-object   v0
    invoke-virtual       Method->invoke(Object, [Object)Object, v0, p2, p4
    move-result-object   v0
    return-object        v0
    move-exception       v0
    const/4              v0, 0
    goto                 :18
    .catch Exception {:0 .. :16} :1A
.end method
```

Figure 12: Smali code of malware to which the API hiding was applied

```
|1|invoke-static args=1 @0x0019{v1, v0, v0, v0, v0}
|1|[fp=0x7fc6064160] Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/
   lang/Class;
|1|ParamCnt : 1, args 1
|1|param1 Ljava/lang/String; android.app.LoadedApk
|1|retval=0x70d46570
|1|return to Lqurfkcv/wzhl/brohq/OLkNShtXcZ;->a(Ljava/lang/String;Ljava/
   lang/String;Ljava/lang/Object;[Ljava/lang/Class;[Ljava/lang/Object;)
   Ljava/lang/Object; [fp=0x7fc6064160]
|1|BYTE_CODE0c00
|1|move-result-object v0
|1|BYTE_CODE6e301b002004
|1|invoke-virtual args=3 @0x001b{v0, v2, v4, v0, v0}
|1|[fp=0x7fc6064160] Ljava/lang/Class;->getMethod(Ljava/lang/String;[Ljava
   /lang/Class;)Ljava/lang/reflect/Method;
|1|ParamCnt : 2, args 3
|1|param1 Ljava/lang/Class; thisptr
|1|param2 Ljava/lang/String; makeApplication
|1|param3 [Ljava/lang/Class; 1893510080
|1|param4 Landroid/app/Application; 319487936
|1|retval=0x130b0340
|1|return to Lqurfkcv/wzhl/brohq/OLkNShtXcZ;->a(Ljava/lang/String;Ljava/
   lang/String;Ljava/lang/Object;[Ljava/lang/Class;[Ljava/lang/Object;)
   Ljava/lang/Object; [fp=0x7fc6064160]
```

Figure 13: Log output when forName, getMethod information is used



```
=================================================================
                      Deobfuscation API Hiding List
=================================================================
[Method] Lqurfkcv/wzhl/brohq/OLkNsHTxCZ;->A(Ljava/lang/String;Ljava/lang/
   String;Ljava/lang/Object[Ljava/lang/)
[Hidden in] Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class
   ;Ljava/lang/reflect/Class;
        ParamCnt : 1, arg 1
        param1 Ljava/lang/String; android.app.LoadeAPK

[Method] Lqurfkcv/wzhl/brohq/OLkNsHTxCZ;->A(Ljava/lang/String;Ljava/lang/
   String;Ljava/lang/Object[Ljava/lang/)
[Hidden in] Ljava/lang/Class;->getMethod(Ljava/lang/String;)Ljava/lang/
   Class;Ljava/lang/reflect/Method;
        ParamCnt : 2, arg 3
        param1 Ljava/lang/Class; thisptr
        param2 Ljava/lang/String; makeApplication
        param3 [Ljava/lang/Class; 1893510080
        param4 Landroid/app/Application; 319487936
```

Figure 14: API Hiding deobfuscation result extracted through ARBDroid

5.3 Can ARBDroid Identify Hidden Sensitive API calls in Real-World Malware?

To evaluate the effectiveness of ARBDroid, we checked what kind of APIs are usually concealed by obfuscation techniques (i.e., class encryption and API hidhing) by using the in-the-wild malware. In this evaluation, we used source and sink APIs as the sensitive APIs [25].

As a result, through the deobfuscation process, we found 1,628 APK that concealed 99,577 sensitive API calls in total as Tab. 2 shows. These APKs were hiding APIs call for obtaining information such as the device status, MAC address of a device, Device ID, and installed packages as in Tab. 3. This evaluation result clearly demonstrates that malicious applications widely use obfuscation techniques to hide their malicious behaviors, and thus, we need automated deobfucators such as ARBDroid to effectively analyze obfuscated malware and to respond with the malware in a timelymanner.

Table 2: Experimental results using the real-world malware to check how many malicious APKs concealed sensitive API calls

The number of malicious APKs hiding sensitive API calls	The number of concealed API calls
1,628	99,577

Table 3: Hidden top API found in malware

Hidden APIs in malware	get Installed Packages	get Parcelable Extra	get Device Id	get Mac Address	get Country	get Subscri berId	Query Intent Activities	Query Intent Services
The number of malicious APKs	1,155	1,134	1,139	1,123	1,058	1,025	960	846

6 Limitations

In the previous section, we discuss the limitations of deobfuscation of the obfuscation techniques, String Encryption, Class Encryption, and API hiding, and how they can be mitigated. The proposed technique is deobfuscating through dynamic analysis, and thus, deobfuscation is impossible for the unexecuted code. To ease the limitation, we employed Android Monkey. Because Android Monkey generates a random event stream, it is possible to test even in unpredictable situations, and it is possible to test without wasting human resources by controlling the frequency of occurrence, cycle of occurrence, and number of occurrences by the user. According to Choudhary et al. [26], Android Monkey achieves higher code coverage than other tools such as SwiftHand [27] and ACTEve [28] and confirmed the highest efficiency. However, we could not overcome the limitation completely. We leave this limitation as future work in which we will aim to achieve the automated execution of Android malware to explore all possible execution paths to deobfuscate obfuscated code in an application as much as possible.

7 Related Work

There are many studies to analyze obfuscated applications and obfuscated malwares. Tiro [29] proposed a framework that can extract Android execution flow through dynamic analysis using the Target Instrument-Run-Observe approach. It can deobfuscate malwares using a combination of existing obfuscation and the latest runtime-based obfuscation technology and analyzes Malicious code with the goal of automatically detecting and deobfuscating language-based and runtime-based obfuscation to enable detection. CopperDroid [30] proposed an automatic VMI-based dynamic analysis system to reconstruct malware behavior. In particular, we presented a method where the analysis of system calls combined with the ability to automatically track and deserialize commonly contextualized IPC and RPC interactions through complex Android objects is key to reconstructing OS and Android-related behaviors. Droidscope [31] presented a precision dynamic binary measurement tool for Android by reconstructing two-level information of the operating system and Java. Both Dalvik bytecode and basic commands are provided to the user as a unified interface that enables dynamic instrumentation, and the interaction of the behavior of Java and components of the malware sample can be checked. Drebin [32], which can detect malwares through static analysis, proposed a method to directly identify malwares on a smartphone. Because limited resources interfere with the monitoring application at runtime, extensive static analysis is performed by collecting as many functions as possible of the application. Contained in the vector space, it automatically identifies common patterns indicative of malwares. In TriggerScope [33], the logic of a malware that is executed or triggered in a specific specified situation is called logic bomb and proposed a new static analysis technique that automatically identifies a trigger to detect logic bomb. and trigger analysis as a static program analysis to identify suspicious trigger conditions protecting potentially sensitive functions by combining dependency analysis by reconstructing and minimizing pathways and overcoming limitations of existing approaches. Andrubis [34] is an automated, large-scale analysis system for Android apps that combine static and dynamic analysis at the Dalvik VM system level. It provides a means to reliably analyze Android apps through tools. Mobile-sandbox [35] is a tool for static and dynamic analysis of malicious behavior in android applications. Static analysis analyzes manifest file syntax and inversely compiles applications. Dynamic analysis executes an application to log all actions performed, including actions initiated from native API calls.

8 Conclusion

In this paper, we proposed ARBDroid that automatically deobfuscates obfuscated Android malware. ARBDroid dynamically analyzes applications to reveal encrypted strings, classes and hidden API calls. We showed that recent malware actively uses obfuscation techniques to conceal malicious behaviors and show that we can effectively analyze obfuscated malware with ARBDroid by using in-the-wild malware. Our evaluation results using real-world malware demonstrate the effectiveness of ARBDroid. We deobfuscated it by using ARBDroid to find concealed information. In our future work, we will focus on the automated execution of Android malware to explore all possible execution paths to deobfuscate obfuscated code in an application as much as possible.

Funding Statement: This work was supported as part of Military Crypto Research Center (UD210027XD) funded by Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. G. C. A. Cimino, N. De. Francesco, F. Mercaldo, A. Santone and G. Vaglini, "Model checking for malicious family detection and phylogenetic analysis in mobile environment," *Computers & Security*, vol. 90, pp. 101691, 2020.

[2] U. Stanley, N. Nkordeh, V. M. Olu and I. Bob-Manuel, "IOT and 5G: The interconnection," *Development*, vol. 1, pp. 2, 2018.

[3] X. Zhang, F. Breitinger, E. Luechinger and S. O'Shaughnessy, "Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations," *Forensic Science International: Digital Investigation*, vol. 39, pp. 301285, 2021.

[4] P. Yan and Y. Zheng, "A survey on dynamic mobile malware detection," *Software Quality Journal*, vol. 26, no. 3, pp. 891–919, 2018.

[5] X. Su, L. Xiao, W. Li, X. Liu, K. Li *et al.,* "DroidPortrait: Android malware portrait construction based on multidimensional behavior analysis," *Applied Sciences*, vol. 10, no. 11, pp. 3978, 2020.

[6] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu *et al.,* "Packergrind: An adaptive unpacking system for android apps," *IEEE Transactions on Software Engineering*, vol. 48, pp. 551–570, 2020.

[7] M. Protsenko and T. Müller, "Protecting android apps against reverse engineering by the use of the native code," in *Int. Conf. on Trust and Privacy in Digital Business*, Berlin, Germany, pp. 99–110, 2015.

[8] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent and T. Le, "Dose: Deobfuscation based on semantic equivalence," in *Proc. of the 8th Software Security, Protection, and Reverse Engineering Workshop*, San Juan, PR, USA. pp. 1–12, 2018.

[9] K. Dillon, "Feature-level malware obfuscation in deep learning," arXiv preprint, arXiv:2002.05517, 2020.

[10] V. Sihag, M. Vardhan and P. Singh, "BLADE: Robust malware detection against obfuscation in android," *Forensic Science International: Digital Investigation*, vol. 38, pp. 301176, 2021.

[11] L. Luo, Y. Fu, D. Wu, S. Zhu and P. Liu, "Repackage-proofing android apps," in *2016 46th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, Toulouse, France. pp. 550–561, 2016.

[12] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor *et al.,* "A large scale investigation of obfuscation use in google play," in *Proc. of the 34th Annual Computer Security Applications Conf.*, San Juan, PR, USA. pp. 222–235, 2018.

[13] Y. Park, T. Park and J. H. Yi, "Multi-partitioned bytecode wrapping scheme for minimizing code exposure on android," *Journal of Internet Technology*, vol. 19, no. 4, pp. 1199–1208, 2018.

[14] M. O. F. K. Russel, S. S. M. M. Rahman, and T. Islam, "A Large-scale investigation to identify the pattern of permissions in obfuscated android malwares," in *Int. Conf. on Cyber Security and Computer Science*, Dhaka, Bangladesh. pp. 85–97, 2020.

[15] J. Kim, N. Go and Y. Park, "A code concealment method using java reflection and dynamic loading in android," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 25, no. 1, pp. 17–30, 2015.

[16] D. Jariwala, "Identification of Malicious Android Applications using Kernel Level System Calls," Ph.D. Thesis, Concordia University, 2014.

[17] H. Rafiq, M. Aleem and M. A. Islam, "On the evaluation of android malware detectors," *Sukkur IBA Journal of Computing and Mathematical Sciences*, vol. 2, no. 1, pp. 20–28, 2018.

[18] A. Cimitile, F. Mercaldo, V. Nardone, A. Santone and C. A. Vis-aggio, "Talos: No more ransomware victims with formal methods," *International Journal of Information Security*, vol. 17, no. 6, pp. 719–738, 2018.

[19] J. Bremer, "Automated analysis and deobfuscation of android apps & malware," *Freelance Security Researcher*, 2013.

[20] C. Bang, J. H. Suk and S. Lee, "VMProtect operation principle analysis and automatic deobfuscation implementation," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 30, no. 4, pp. 605–616, 2020.

[21] J. Lee, J. Han, M. Lee, J. Choi, H. Baek *et al.,* "A study on api wrapping in themida and unpacking technique," *Journal of the Korea Institute of Information Security & Cryptology*, vol. 27, no. 1, pp. 67–77, 2017.

[22] H. Cho, J. Lim, H. Kim and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on android platform," *Journal of Supercomputing*, vol. 72, no. 1, pp. 232–246, 2016.

[23] S. Millar, N. McLaughlin, J. M. D. Rincon, P. Miller and Z. Zhao, "DANdroid: A multi-view discriminative adversarial network for obfuscated android malware detection," *Tenth ACM Conference on Data and Application Security and Privacy*, pp. 353–364, 2020.

[24] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel *et al.,* "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[25] P. Patel, G. Srinivasan, S. Rahaman and I. Neamtiu, "On the effectiveness of random testing for android: or how i learned to stop worrying and love the monkey," in *Proc. of the 13th Int. Workshop on Automation of Software Test*, Gothenburg, Sweden. pp. 34–37, 2018.

[26] S. R. Choudhary, A. Gorla and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *IEEE/ACM Int. Conf. on Automated Software Engineering*, Lincoln, NE, USA. pp. 429–440, 2015.

[27] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[28] J. Qin, H. Zhang, S. Wang, Z. Geng, and T. Chen, "Acteve++: An improved android application automatic tester based on acteve," *IEEE Access*, vol. 7, pp. 31358–31363, 2019.

[29] M. Y. Wong, Y. Michelle, and D. Lie, "Tackling runtime-based obfuscation in android with {TIRO}," in *USENIX Security Symp.*, Baltimore, MD, USA. pp. 1247–1262, 2018.

[30] K. Tam, S. J. Khan, A. Fattori and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," *Ndss*, pp. 1–15, 2015.

[31] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the {OS}and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symp.*, Bellevue, WA. pp. 569–584, 2012.

[32] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck *et al.,* "Drebin: Effective and explainable detection of android malware in your pocket," *Ndss*, vol. 14, pp. 23–26, 2014.

[33] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel *et al.,* "Triggerscope: Towards detecting logic bombs in android applications," in *IEEE Symp. on Security and Privacy*, San Jose, CA, USA. pp. 377–396, 2016.

[34] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," *International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pp. 3–17, 2014.

[35] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp and J. Hoffmann, "Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques," *International Journal of Information Security*, vol. 14, no. 2, pp. 141–153, 2015.