String Deobfuscation Scheme based on Dynamic Code Extraction for Mobile Malwares

WooJong Yoo, Myeongju Ji, MinKoo Kang, and Jeong Hyun Yi*
Soongsil University, Seoul, 06978, Korea
{msecwj, wlaudwn007, minku1024}@gmail.com, jhyi@ssu.ac.kr

Abstract

Various code protection schemes are being implemented to offer protection and address the vulnerabilities of Android applications. Among these schemes, code obfuscation is widely used. Because the program code string is especially fundamental in identifying methods or variables, if this information is exposed to the attacker, reverse engineering analysis becomes that much easier. Consequently, string encryption should be prioritized among obfuscation techniques. If malware rather than normal code happens to misuse such a protection function, it can actually bring about an opposite effect by making code analysis more difficult. Hence, in this paper, we first propose a string deobfuscation scheme that, to effectively analyze malware that uses string encryption, acquires the decrypted string through dynamic code extraction and then introduce the system design and implementation.

Keywords: Deobfuscation, Malwares, Reverse Engineering

1 Introduction

As the Android smart phone market grows, the number of applications that provide users convenience or various enjoyments is also exponentially growing. However, because of structural issues for Android, these applications are exposed to many security threats and the number of malware that abuses this feature is also sharply increasing [6]. Among several application protection schemes, code obfuscation is commonly used by many [8]. Code obfuscation, however, is not only used for the purpose of protecting the application, but malware developers are increasingly abusing this tool to ensure that their code is not exposed to the anti-virus tool [7]. Because code analysis for malware with obfuscation applied takes longer, in the case of a zero-day attack, it takes too long to counteract, proving it difficult to effectively resolve the problem. In this case, the obfuscated code first needs to be converted into deobfuscated code, and the current method depends on the experience of the analyzer as he or she identifies the obfuscation method applied on the code and then manually works on it. To analyze code using such a procedure, one must know the characteristics of each of the various obfuscation tools, meaning that with the advent of each new tool, one must repeat the cumbersome work of having to research and add to one's expertise. For example, the *dex-oracle* [1] tool is a deobfuscator which is dedicated to DexGuard. It generates a pattern of the instruction corresponding to the string encryption algorithm by using regular expression. The detailed operations of the *dex-oracle* tool are as follows. It firtst decompiles applications to obtain the smali code, searches for the relevant algorithm regarding to the string encryption, and invokes the corresponding decryption method through the emulator. And that, the readable strings resulted from the decryption method are revealed. Although the *dex-oracle* tool is applicable to deobfuscate the obfuscated strings only with DexGuard, it does not works for any string encryption generated by other obfuscation tools other than DexGuard. To solve these problems, we need a function that will automatically deobfuscate the code without needing the involvement of the analyzer. Particularly, it is most important

*Corresponding author: School of Software, Soongsil University, Seoul, 06978, Korea, Tel: +82-2-828-7360 that among the identifiers such as variables, function name, etc., the must fundamentally applied string encryption function is removed. Thus, in this paper we propose a string deobfuscation scheme that will automatically decrypt on the platform level the obfuscated malware with string encryption without prior knowledge of the encryption key.

2 Proposed Scheme

In this section, we explain the string deobfuscation scheme that, through dynamic code extraction of the application, acquires the original string using the return value of the decryption method without prior knowledge of the encryption key.

2.1 Approach

To decrypt the encrypted string, one can use the method of extracting the encryption key by discerning the principles regarding the algorithm used in the encryption and then decrypting the key. However, there are countless commercial obfuscation tools, and it is unrealistic to analyze and counteract each algorithm used in every tool. As a result, we need a generic method that, unlike existing methods where the analyzer gets involved and manually analyzes each particular obfuscator, will automatically distinguish the encrypted string and decrypt it without needing prior knowledge on the specific encryption algorithm.

To make generic decryption possible, we must first focus on decryption methods and their results. Regardless of which encryption algorithm is used, the encrypted string must be decrypted into human-readable string for it to be used as symbols such as variables and function name within the application. For this to happen, in the case of applications with string encryption, the program must include a decryption method or logic. In addition, because the result of decryption will be the original string, it is important to find out the location where the execution result of the decryption method is sent.

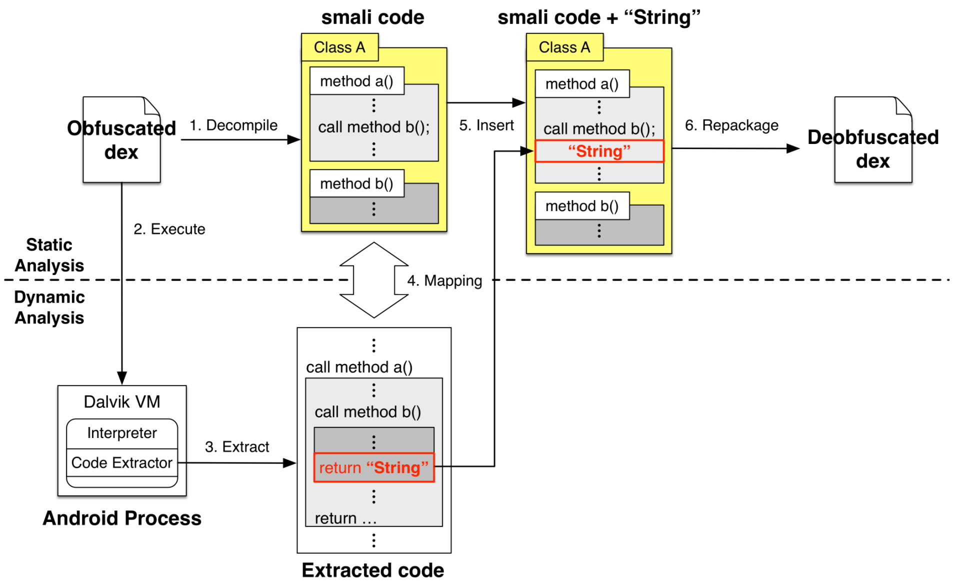

Figure 1: Proposed Scheme for String Deobfuscation on Android

In this paper, we propose a method where deobfuscation is always possible even without the encryption key by modifying Dalvik VM to dynamically extract the codes that are activated when the application is real-time running and then, finally, injecting the decrypted string into the existing application and repackaging it. This process is explained in more detail below (See Figure 1). First, we decompile the obfuscated dex, which is the target application, and obtain the smali code. This statically obtained code contains the logic of the methods used in the target application. Next, we run the target application in the Dalvik virtual machine and dynamically extract the bytecode of the obfuscated dex. At this point, the extracted code, unlike the smali code obtained through static analysis, contains not only the logic of the methods actually run but also the return values. After sorting out among the return values acquired in this way those whose data type is string, we map the extracted code and decompiled code and identify the corresponding smali code parts for same methods. Then, after inserting the return string obtained from the extracted code into the return point of the decompiled code of that method, the application is repackaged. In this way, although all the methods that return to string type are not decryption methods, a decryption method is certainly included so that it is neutralized from string encryption obfuscation.

2.2 Design

The proposed `String Deobfuscator`, as shown in Figure 2, is largely comprised of `Code Extractor`, `Extracted Code Parser`, `Dex-Extracted Code Mapper`, and `Smali Rewriter`. In the following section, we give a brief description of each component and its function

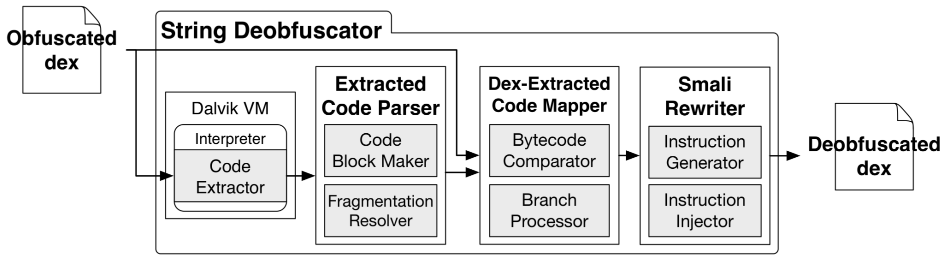

Figure 2: Architectural Design of String Deobfuscator

2.2.1 Code Extractor

Embedded within Dalvik VM, this module dynamically extracts the bytecode that is executed by the interpreter. `Code Extractor` outputs codes such as the bytecode and smali code of the currently run instruction as well as dynamic information such as variables, branch offset, etc. into text format.

2.2.2 Extracted Code Parser

`Extracted Code Parser`, as the module that parses the dynamically extracted code from the `Code Extractor`, is composed of the code block maker and fragmentation resolver module. Code Block Maker splits the extracted code by instruction and generates code blocks. A code block is made of code and dynamic information, and the decrypted string needed for the proposed scheme is in the dynamic information of the 'return' type instruction code blocks. In addition, the code and dynamic information of the extracted code is generally sequentially output according to the order of the dex instruction. However, in the case of Java's static block, there have been instances where a different instruction code block was inserted and output among dynamic information. When a code block fragmentation like this occurs, the fragmented code block needs to be consolidated into one code block, which is the job of the `Fragmentation Resolver`.

2.2.3 Dex-Extracted Code Mapper

`Dex-Extracted Code Mapper` is responsible for mapping the extracted code and smali code. Using the extracted code as the standard, bytecode is compared in units of instruction following along the actual executed control flow. Through code mapping, we find the location to insert the decrypted original string. Taking a more detailed look at the functions, they can be split into bytecode comparator and branch processor. The bytecode comparator sequentially compares the bytecode of the smali code acquired by decompiling the dex and bytecode of the code block generated through the extracted code parser. Using the first function instruction in the extracted code, the comparison location for smali code is chosen. It finds the calling function using the method descriptor, which includes the class name, method name, return type, and parameter, within the dynamic information of the function call instruction and begins comparisons. Likewise, in the case of a function call or an exception, it finds the location of the smali code to be compared and compares it. In the case that the location of the instruction to be compared with is changed into function units, the return point is saved. Furthermore, while most instructions can be sequentially compared, in the case of branch instructions such as ‘if’ or ‘goto’, after the comparison, the location of the next instruction to be compared must be found and changed. In the case of branch instructions such as ‘if’ or ‘goto’, the `Branch Processor` confirms offset using the dynamic information of the instruction and takes care of the instruction after the offset.

2.2.4 Smali Rewriter

`Smali Rewriter` ensures that after determining the insertion location of the decrypted original string found through code mapping, the decrypted string is added into the smali code and the dex file is rewritten. It recompiles the rewritten smali code and lastly generates the deobfuscated dex.

3 Experimental Result

`Code Extractor` was built by modifying the interpreter source code for Android version 4.4.4 while the `Extracted Code Parser`, `Dex-Extracted Code Mapper`, and `Smali Rewriter` were built using Java (JDK 1.7) in Windows 7.

In advance, sample applications were applied with string encryption of commercial obfuscation tools and then proceeded debofuscation experiments. Allatori [2], DexGuard [3], and DexProtector [4] were selected for commercial obfuscation tools. Strings that needs protections are replaced into unreadable strings when an application is applied with string encryption of Allatori.

In Figure 3, the red of the smali codes are the unreadable strings replaced by Allatori. In Figure 4 shows the extracted code obtained from `Code Extractor` and can be observed that the decryption method(blue) returns decrypted string(red). Deobfuscation is completed by inserting the decrypted string the the application via the `String Deobfuscator`. Figure 5 is the smali code of the deobfuscated application.

Obfuscation tools such as DexProtector and DexGuard, uses different decryption methods and string encryption algorithms but was able to extract decrypted strings.

Then, malware applied with string encryption was targeted to extract hidden strings and deobfuscate. Malwares were obtained from Android PRAGuard Dataset [5]. Figure 8 is smali code using URL for malicious behavior and was extracted by `Code Extractor`. Figure 9 is malware’s deobfuscated smali code. Analysts can easily identify which classes and methods have malicious behavior by inserting deobfuscated strings.

```
# direct methods
.method public constructor <init>(Lcom/example/msecdeobfuscatortest/MainActivity;)V
   const/4 v0, 0x0
    const-string v1, "t\'z1 0k 13&&n:"
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    invoke-static {v1}, Lcom/example/msecdeobfuscatortest/x;->A(Ljava/lang/String;)Ljava/lang/String;
    move-result-object v1
    iput-object v1, p0, Lcom/example/msecdeobfuscatortest/n;->D:Ljava/lang/String;
    iput-object v0, p0, Lcom/example/msecdeobfuscatortest/n;->a:Lcom/example/msecdeobfuscatortest/
MainActivity;
    iput-object p1, p0, Lcom/example/msecdeobfuscatortest/n;->a:Lcom/example/msecdeobfuscatortest/
MainActivity;
    return-void
.end method
```

Figure 3: Smali code obfuscated by Allatori

```
…
|1|invoke-static args=1 @0x1eb7 {v1, v0, v0, v0, v0}
|1|[fp=0x41976d80] Lcom/example/msecdeobfuscatortest/x;->A(Ljava/lang/String;)Ljava/lang/String;
…
|1|BYTE_CODE1100
|1|return-object v0
|1|retStr = decrypted string of Allatori
|1|return to Lcom/example/msecdeobfuscatortest/n;-><init>(Lcom/example/msecdeobfuscatortest/
MainActivity;)V [fp=0x41976db8]
…
```

Figure 4: Extracted code deobfuscated by Allatori

```
# direct methods
.method public constructor <init>(Lcom/example/msecdeobfuscatortest/MainActivity;)V
    const/4 v0, 0x0
    const-string v1, "t\'z1 0k 13&&n:"
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    invoke-static {v1}, Lcom/example/msecdeobfuscatortest/x;->A(Ljava/lang/String;)Ljava/lang/String;
    move-result-object v1
    goto :goto_d
    const-string v0, "[MSEC] Decrypted String : decrypted string of Allatori"
    :goto_d
    iput-object v1, p0, Lcom/example/msecdeobfuscatortest/n;->D:Ljava/lang/String;
    iput-object v0, p0, Lcom/example/msecdeobfuscatortest/n;->a:Lcom/example/msecdeobfuscatortest/
MainActivity;
    iput-object p1, p0, Lcom/example/msecdeobfuscatortest/n;->a:Lcom/example/msecdeobfuscatortest/
MainActivity;
    return-void
.end method
```

Figure 5: Smali code deobfuscated by Allatori

```
…
|1|BYTE_CODE713016000000
|1|invoke-static args=3 @0x0016 {v0, v0, v0, v0, v0}
|1|[fp=0x41976d80] Lo/if;->´(III)Ljava/lang/String;
|1|BYTE_CODEda070702
|1|mul-int/lit8 v7,v7,#+0x02
…
|1|BYTE_CODE1100
|1|return-object v0
|1|retStr = decrypted string of DexGuard
|1|return to Lo/if;-><init>(Lcom/example/msec/MainActivity;)V [fp=0x41976db8]
…
```

Figure 6: Extracted code obfuscated by DexGuard

The deobfuscation experiments were conducted on applications applied with various string encryption, and was able to extract deobfuscated strings on each applications without analyzing the encryption algorithm and key.

```
…
|1|BYTE_CODE71101c000000
|1|invoke-static args=1 @0x001c {v0, v0, v0, v0, v0}
|1|[fp=0x41976d38] Lcom/example/msec/SecretClass;->secretMethod(Ljava/lang/String;)Ljava/lang/String;
…
|1|BYTE_CODE1102
|1|return-object v2
|1|retStr = decrypted string of DexProtector
|1|return to Lcom/example/msec/SecretClass;-><init>(Lcom/example/msec/MainActivity;)V [fp=0x41976db8]
…
```

Figure 7: Extracted code obfuscated by DexProtector

```
…
|1|BYTE_CODE703022001002
|1|invoke-direct args=3 @0x0022 {v0, v1, v2, v0, v0}
|1|[fp=0x41976c54] Ljava/lang/String;-><init>([BI)V
|1|retval=0xe463451642b24238
|1|return to Ljp/oomosirodougamatome/MainActivity;->$(III)Ljava/lang/String; [fp=0x41976c7c]
|1|BYTE_CODE1100
|1|return-object v0
|1|retStr = http://depot.bulks.jp/get41.php
|1|return to Ljp/oomosirodougamatome/MainActivity;-><init>()V [fp=0x41976cb4]
…
```

Figure 8: Extracted code of malware

```
.class public Ljp/oomosirodougamatome/MainActivity;
…
.method public constructor <init>()V
…
    invoke-static {v1, v2, v0}, Ljp/oomosirodougamatome/MainActivity;->$(III)Ljava/lang/String;
    move-result-object v0
    goto :goto_14
    const-string v0, "[Decrypted String] : http://depot.bulks.jp/get41.php"
    :goto_14
    iput-object v0, p0, Ljp/oomosirodougamatome/MainActivity;->url:Ljava/lang/String;
    const-string v0, ""
    iput-object v0, p0, Ljp/oomosirodougamatome/MainActivity;->androidid:Ljava/lang/String;
…
.end method
```

Figure 9: Deobfuscated smali code of malware

4 Conclusion

Due to the structure of Androids, there has been a steep increase in the number of malware. There has also been a trend of an increasing number of cases where malware developers will apply obfuscation to ensure that their personal code is not exposed to the anti-virus tool. Because code analysis for malware with obfuscation takes a lot of time, it is difficult to effectively counteract.

Thus, to target malware that uses string encryption obfuscation, we proposed in this paper a string deobfuscation scheme that can automatically decrypt at the platform level without prior knowledge of the encryption key and, through experiments, were able to accurately verify the scheme's workings. We anticipate that with as many countless malware are abusing obfuscation, the proposed scheme can be that much more effectively utilized in malware analysis, etc.

Acknowledgments

This research was supported by the Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029).

References

[1] dex-oracle `https://github.com/CalebFenton/dex-oracle`.

[2] Allatori `http://www.allatori.com/`.

[3] DexGuard `https://www.guardsquare.com/dexguard`.

[4] DexProtector `https://dexprotector.com/`.

[5] PRAGuard `http://pralab.diee.unica.it/en/AndroidPRAGuardDataset`.

[6] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[7] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[8] P. Schulz. Code protection in android. *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany*, 110, 2012.

Author Biography

WooJong Yoo received the B.S. degree in Computer Science and Engineering from Soongsil University in 2015. Currently he is taking a master's course at Graduate School of Computer Science and Engineering, Soongsil University. His research interests include Android, Mobile Security and Deobfuscation.

Myeongju Ji received the B.S. degree in Computer Science and Engineering from Soongsil University in 2014. Currently he is taking a master's course at Graduate School of Computer Science and Engineering, Soongsil University. His research interests include Android, Mobile Security and Deobfuscation.

MinKoo Kang received the B.S. degree in Computer Science and Engineering from Soongsil University in 2016. Currently he is taking a master's course at Graduate School of Computer Science and Engineering, Soongsil University. His research interests include Android, Mobile Security and Deobfuscation.

Jeong Hyun Yi is an Associate Professor in the School of Software and a Director of Mobile Security Research Center at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, IoT security, and applied cryptography.