Mobile Code Packing Scheme Based on Multi-partitioned Bytecode Wrapping

Yongjin Park[1], Taeyong Park[1], Sung Tae Kim[2], and Jeong Hyun Yi[1]*
[1]School of Computer Science and Engineering, Soongsil University, Seoul, 06978, Korea
{absolujin, taeyong88}@gmail.com, jhyi@ssu.ac.kr
[2]Graduate School of Software, Soongsil University, Seoul, 06978, Korea
setmenuda1@gmail.com

Abstract

Android apps are structurally more vulnerable to reverse engineering attacks relative to other mobile apps. Several different methods are being employed to protect apps from such attacks, and among those is packing. However, even with packing, a problem exists where the attacker can obtain the entire original bytecode through dynamic analysis. As a result, in this paper, we propose a scheme to improve existing methods by addressing the problem of exposing all of the original bytecode by first splitting the original bytecode into multiple parts and then wrapping and dropping them to prevent reverse engineering analysis. Furthermore, we constructed the proposed scheme and compared its performance and level of security with alternative existing methods.

Keywords: Android, Mobile Application, Security

1 Introduction

Best represented by Google Android and Apple iOS, the smartphone functions not only as a communication device like the phone or internet but offers a variety of services through the installment of applications desired by the user. The utilization of the smartphone to store personal information as well as for financial services is increasing. In the case of Androids who have the largest number of users (82.8%) [10], due to structural problems, reverse engineering attacks are not only easily launched, but the attacker can self-sign and repackage apps [7]. To protect vulnerable Android apps from such attacks, developers apply code obfuscation or tamper detection methods. Also, recently, packing methods centered on commercial protection services such as DexProtector [2], Bangle [1], Ijiami [3] that protect code through encryption are being employed. Packing is a method that, to prevent malware from running static analysis in the existing PC environment [12], increases static analysis resistibility [5] through executable compression such as by UPX [4] by using the characteristics of the file format or loader.
However, because the packing methods provided by current packers load the entire original bytecode onto the memory while the app loads, the original bytecode can be easily extracted and analyzed in dynamic analysis environments such as DECAF [8], DexHunter [14], or AppSpear [13], proving it difficult to effectively protect the original bytecode.
Currently the code by reducing exposure in recent studies of PC-based code protection [6] [9] has developed into a form of protection to ensure that attacker can not prevent access to the code, or know the location of the code. Research is needed to protect the code by reducing exposure from attacker to code efficiently in mobile protection.
In this paper, we propose a method that addresses the problem of existing packers exposing all of the

Research Briefs on Information & Communication Technology Evolution (ReBICTE), Vol. 2, Article No. 12 (September 15, 2016)
*Corresponding author: School of Computer Science and Engineering, Soongsil University, Seoul, 06978, Korea, Tel: +82-2-820-0914

original bytecode by minimizing the exposure of the original bytecode in the memory using wrapping and dropping to effectively increase the resistibility against dynamic analysis. The proposed method splits the original bytecode into several codes and wraps them so that the wrapped code is only unwrapped when it is called and then loaded onto the memory. Once that code is used, it is dropped from the memory in order to minimize the time and amount of exposure of the original bytecode in the memory.

2 Proposed Scheme

While packing methods can vary in their detailed procedures or construction depending on the device, the fundamental theory behind the procedure of packing an app and unpacking and then running a packed app is shown in Figure 1. The process of applying packing entails packing the original bytecode using methods such as encryption and then placing the wrapped bytecode within the app. Code that will unpack the original bytecode is added, and the entry point of the app is changed to the unpacking code. When the app is run, the unpacking code designated as the entry point is executed. Lastly, once the original bytecode is unpacked and loaded onto the memory, the original bytecode is called [11].

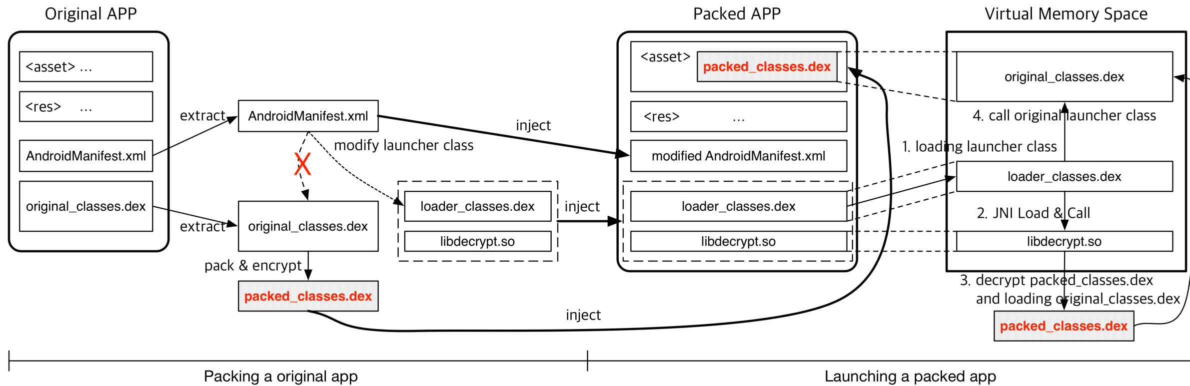

Figure 1: Operation procedures of existing packing schemes

As mentioned before, the packing methods of existing packers pack the entire original bytecode and then, during the app runtime, unpack the entire packed code before loading it onto the memory. This causes the entire original bytecode to be exposed in the memory from the time of launching the app until its termination, resultingly lowering resistance against dynamic analysis. Thus, in this paper, we propose a scheme that utilizes wrapping and dropping to prevent the exposure of the entire original bytecode in the memory.

Instead of packing the entire code, the proposed scheme protects the original bytecode by splitting the core bytecode to decrease the size and time of exposure of the original bytecode. To do this, the core bytecode is composed of more than one method or class, and the multiple core bytecodes are wrapped separately to limit its exposure. Furthermore, the core bytecode is only exposed from when it is called to when it is executed to minimize the time the code is exposed. Wrapping and dropping, the core functions of the proposed method, can be explained as follows.

2.1 Wrapping

Wrapping is a technique that encrypts into binary form and hides the core bytecode that has been sorted into units of class or method within the native module and then, after decrypting those, calls them using stub code. Figure 2 illustrates the process of wrapping the core bytecode that has been sorted from the

original bytecode and adding them to the module.

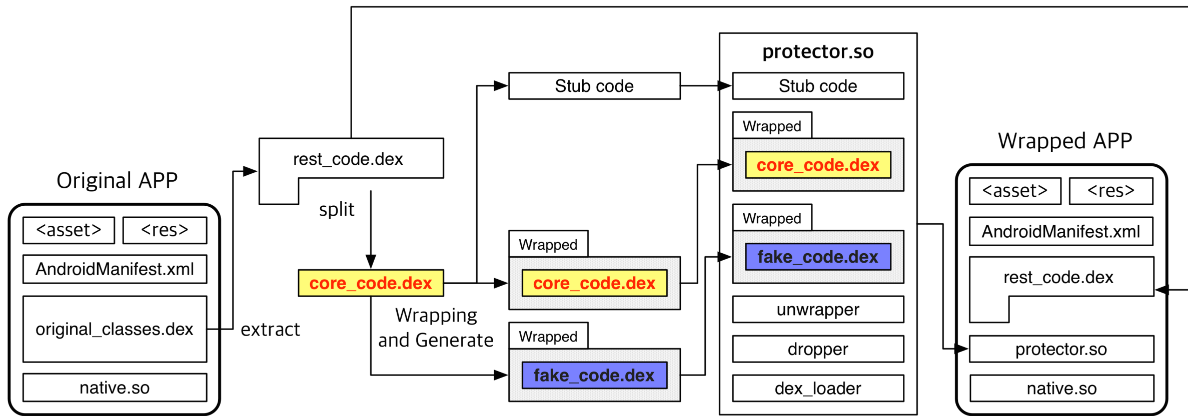

Figure 2: Wrapping process of the proposed scheme

The stub code that can call the core bytecode, sorted from the original bytecode, and the fake bytecode used for dropping are all generated. Both the stub code and fake bytecode are generated dependent on the signature of the core bytecode.
Both the core bytecode and fake bytecode is wrapped through encryption and inserted into the protection module along with the stub code. The protection module is now composed of not only the added code but also the unwrapper, dropper, and DEX loader and is generated into the native module and included in the app. Lastly, the remaining bytecode excluding the core bytecode is regenerated and included in the app.
If the core bytecode is called after the app is executed, the bytecode is unwrapped and loaded onto the memory through a process shown in Figure 3 below.

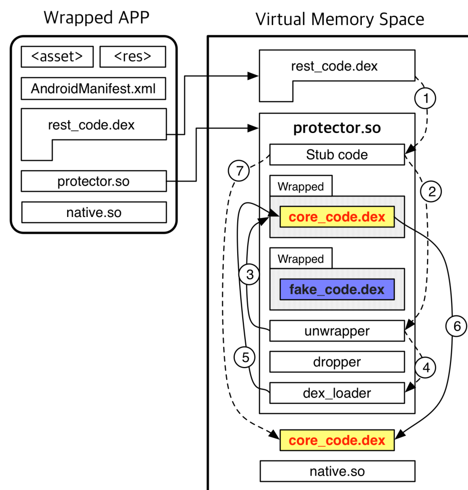

Figure 3: Unwrapping process of the proposed scheme

The first call bytecode has the native protection module load using a JNI call, and the call is connected to the stub code. The stub code first calls the unwrapper and unwraps the wrapped core bytecode. Secondly, the DEX loader is called and the unwrapped core bytecode is loaded onto the memory. Lastly, the core bytecode loaded during the stub code is called.

Through the process mentioned above, the amount of exposure in the memory is minimized. In addition, the core bytecode is encrypted within the native protection module and included in binary form and, by calling using JNI, there are additional benefits that are identified as follows.
To dynamic load bytecode in Android apps, the framework's DexClassLoader class is generally used. When this API is used, it becomes easier to dynamic load wrapped core bytecode and to identify the call code. If discerning core bytecode with protection schemes applied becomes easier, the analysis time decreases, benefitting the attacker. Consequently, to ensure that identification is not made easy, we included DEX Loader, which is a native dynamic loading module that uses `dvm_dalvik_system_DexFile`, the native API of Dalvik VM.
The call bytecode is made to call the wrapped core bytecode using JNI call and, after the unwrapper causes the wrapped bytecode to be unwrapped, the core bytecode is dynamic loaded onto the memory using the DEX loader. Because the JNI call is a frequently used call in Android apps to ensure the reusability and performance of the native module, identifying the call used for the proposed scheme becomes difficult and can delay the analysis time for the attacker.

2.2 Dropping

Dropping is a technique included to address the existing packers' exposure of the original bytecode until the app is terminated. Similar to how with wrapping the bytecode is only loaded when the selected core bytecode is called, bytecode is individually unloaded from the memory after it is used.

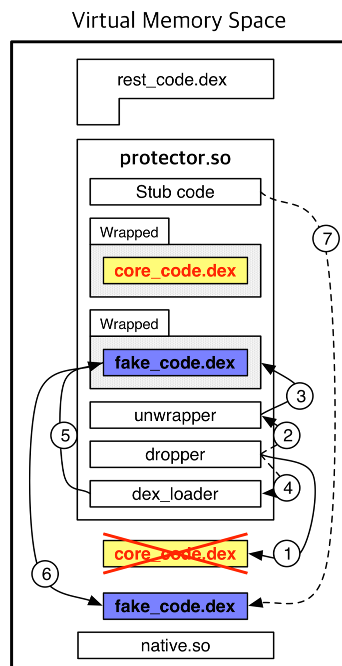

Figure 4: Dropping process of the proposed scheme

As shown in Figure 4, dropping is used to minimize exposure time by having core bytecode removed from the memory once it is processed. In addition, after loading the fake bytecode that was included when wrapping, it is connected to the stub code. This fake bytecode has a similar signature as the core bytecode but contains completely different logic, causing confusion for the attacker during dynamic analysis.

3 Experimental Results

Figure 5 and Figure 6 shows the content of the core bytecode and fake bytecode of an app that has applied the proposed scheme. In addition, to check exposure of core bytecode in the memory, we dump the memory after the core bytecode is executed as shown in Figure 7. We can see that after the core bytecode was run, the core bytecode was dropped from the memory and removed, and the fake bytecode was dumped instead.

```
root@hammerhead:/data/andromonitor # ./busybox hexdump /data/temp/libobfuscated.so -x -s 0x43FC -n 22
00043fc    0012    011a    000a    206e    0003    0012    010a    0138
000440c    0003    1012    000f
```

Figure 5: File dump of the core bytecode file

```
root@hammerhead:/data/andromonitor # ./busybox hexdump /data/temp/libobfuscated.so -x -s 0x4160 -n 22
0004160    1212    0012    1112    1037    0003    1012    2037    0003
0004170    020f    0212    fe28
```

Figure 6: File dump of the fake bytecode file

```
(gdb) x/11hx 0x77101158
x/11hx 0x77101158
0x77101158:     0x1212  0x0012  0x1112  0x1037  0x0003  0x1012  0x2037  0x0003
0x77101168:     0x020f  0x0212  0xfe28
```

Figure 7: memory dump after running the core bytecodes

Table 1 shows the comparison of the proposed schemes with current packers regarding exposure time and size of the original bytecode. In the case of apps using the proposed scheme, the original bytecode is only exposed from when the code is called until when the code is done running, and the amount of exposure is minimized as well. Furthermore, due to the fake bytecode, it is difficult to identify the core bytecode in the memory.

In addition, to measure the performance targets NotePadSerial.apk app from the Nexus 5 (Android 4.4.4) instrument to measure performance. As a result of measuring a performance, current packers delay (about 150-230 msec) is generated as the unpacking process performed at the time of launching the app. On the other hand, the proposed scheme it take for no difference or shorter time of the original app (338 msec) when the app launch. In core bytecode execution performance case, packers do not have much dealy as compared to the original app (0.013 msec), but the proposed scheme is to delay (about 13 msec) is increased each time you call the core bytecode. But because of delays in launching the app significantly smaller than packers, proposed scheme can be said to be a performance superior to packers.

	DexProtector	Bangle	Ijiami	Proposed Scheme
Number of Packed Code	Multiple	Single	Single	Same as core bytecode or more
Identifying Code on Memory	Easy	Easy	Hard	Hard and temporary exposure
Code Visibility on Memory	Entire DEX	Entire DEX	Entire DEX	Only working code
Unpakcing Time	App launching	App launching	App launching	Calling core bytecode
Code Lifetime	Same as app	Same as app	Same as app	While core bytecode running
App Launching Time(msec)	608	660	487	354
Core bytecode Execution Time(msec)	0.020	0.011	0.010	13.123

Table 1: Feature and performance comparison of the proposed scheme

4 Conclusion

Among existing mobile app protection methods, the packing methods provided by commercial services seem to offer effective resistivity against static analysis. However, with research being done surrounding the dynamic analysis environment lately, existing packing methods are no longer sufficient to provide enough resistance against dynamic analysis. The reason being that existing packing methods unpack the entire original bytecode from the time the app launches and keep them in the memory until the time the app is terminated.
To address these shortcomings, apps that apply the proposed scheme have the core bytecode split from the original bytecode and grouped in units of method or class and wrapped and after the core bytecode is run, they are removed from the memory through dropping, thus effectively decreasing the size and amount of exposure. These proceduresin the proposed scheme ensure greater resistance against dynamic analysis compared to existing packing methods and performace is also excellent.

Acknowledgments

This research was supported in part 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] Bangcle. `http://www.bangcle.com`.

[2] Dexprotector. `http://www.dexprotector.com`.

[3] Ijiami. `http://www.ijiami.cn`.

[4] Upx. `http://upx.sourceforge.net`.

[5] We can still crack you! general unpacking method for android packer (no root). In *Proc. of the Black Hat Briefings (BLACKHAT'15), Marina Bay Sands, Singapore*, pages 1–1, March 2015.

[6] S. Crane, C. Liebchen, A. Homescu, and L. Davi. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. of the 36th IEEE Symposium on Security and Privacy (SSP'15), San Jose, California, USA*, pages 763–780. IEEE, May 2015.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. of the 20th USENIX conference on Security (SEC'11), San Francisco, California, USA*, pages 135–147. ACM Press, August 2011.

[8] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proc. of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14), San Jose, California, USA*, pages 248–258. ACM Press, July 2014.

[9] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proc. of the 22nd ACM Conference on Computer and Communications Security (CCS'15), Denver, Colorado, USA*, pages 280–291. ACM Press, October 2015.

[10] I. Research. Smartphone os market share, 2015 q2. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, 2015.

[11] T. Strazzere and J. Sawyer. Android hacker protection level 0. `https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDATED.pdf`, October 2014.

[12] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. of the 36th IEEE Symposium on Security and Privacy (SSP'15), San Jose, California, USA*, pages 659–673. IEEE, May 2015.

[13] 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 *Proc. of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15), Kyoto, Japan, LNCS*, volume 9404, pages 359–381. Springer-Verlag, November 2015.

[14] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Proc. of the 20th European Symposium on Research in Computer Security (ESORICS'15), Vienna, Austria, LNCS*, volume 9327, pages 293–311. Springer-Verlag, September 2015.

Author Biography

Yongjin Park received the B.S degree in Computer Science and Engineering from Soongsil University in 2014. M.S. degrees in Computer Science and Engineering from Soongsil University in 2016. Currently he is a research staff in Mobile Security Research Center. His research interests include Mobile Security, Android Platform and IoT.

Taeyong Park 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, Soongsil University. His research interests include Mobile Security, Obfuscation and System Programming.

Sung Tae Kim received the B.S. degree in Electronic Engineering from Soongsil University in 2015. Currently he is taking a master's course at Graduate School of Software, Soongsil University. His research interests include Mobile Security and Android Platform.

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.