DynVMDroid: Android App Protection via Code Disorder and Dynamic Recovery

Weimiao Feng[1], Rui Hu[1,2](✉), Cong Zhou[1,2], and Lei Yu[1]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{fengweimiao,hurui,zhoucong,yulei1993}@iie.ac.cn

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract. To protect Android applications from reverse engineering, more and more adversarial analysis techniques are proposed, such as packing, encryption, obfuscation, etc. As one of the most advanced techniques for obfuscation, code virtualization at the dex bytecode level has evolved from hiding meta information to protect executable instructions. However, previous approaches are proved to have a certain degree of vulnerability at the directive opcode replacement. In this paper, we present DynVMDroid, a reinforcement system based on code virtualization to protect Android applications from reverse engineering. DynVMDroid consists of two components, a reinforcement engine and a custom runtime environment. The reinforcement engine disrupts the inherent structural order and extends the length of the original instructions from key methods, converting them into virtual code in Android applications. The custom runtime environment dynamically recovering the virtual instructions to ensure the protected application work properly. To verify its performance and compatibility, we have applied DynVMDroid to 10 applications. In addition, various attack methods have been adopted on the protected applications to validate their security. Our experimental results show that the applications protected by DynVMDroid perform correctly and effectively against common reverse analysis techniques with acceptable performance losses.

Keywords: Android application reinforcement · Code disorder · Dynamic recovery

1 Introduction

With the popularity of Android, many users take it as their first choice and download various applications from the app store. Unfortunately, some unauthorized reverse engineers crack the applications in order to profit from them. This worrisome phenomenon leads to threats to users' privacy and infringement of developers' intellectual property rights. So far, many solutions to reinforce Android applications rely on mature technologies such as dynamic loading, encryption,

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 571, pp. 244–257, 2024.
https://doi.org/10.1007/978-3-031-56583-0_16

anti-debugging, etc., which means that some reverse analysts with special purposes also understand these techniques and develop countermeasures. A class of schemes known as code obfuscation [2,4,6–8,11–13,16] makes reverse engineering harder. Among them, code virtualization based on virtual machine (VM) hides real code by converting original instructions into virtual instructions, greatly raising the threshold for reverse engineering. However, recent research [?] shows that an analyst can analyze the virtual codes through the specific instruction format.

In this paper, we propose DynVMDroid, an Android application reinforcement system based on code disorder and dynamic recovery to combat this attack mode. It implements high-intensity protection, preventing Android applications from reverse engineering. Our key insight is that if the Dalvik instruction format is altered, then it will be hard for analysts to analyze the logic of the program based on the inherent structure, and the ordinary attack approach will no longer work. Different from the previous work, which only maps the opcodes of instructions while the arrangement of virtual opcodes and operands still maintain the original format, we disorder the opcode and operand in each instruction and implement a custom interpreter to dynamically recover the instructions. Our carefully designed scheme guarantees the consistency of program operation results although the instructions look different from analysts' perspective. To analyze the applications protected by the DynVMDroid system, they are forced to analyze every single executed instruction, which greatly increases the cost of reverse engineering the code.

We conduct experiments to ensure the applications reinforced by DynVMDroid is compatible with popular Android platforms without functions change. The security test includes manual attack and universal deobfuscating tool. Besides, we employ the performance test on protected applications. The experimental results show that the applications reinforced by DynVMDroid have certain security and the performance indicators are within an acceptable range.

There are some challenges in our work. First, code virtualization at Java level is not widely used due to efficiency and compatibility problems. Our work needs to ensure the protected applications work correctly with modest cost. Second, the Dalvik interpreter interprets instructions according to specific instruction format while our designed system breaks this rule by changing the original Dalvik instruction format. Therefore, we need to consider the influence of the transformation on instructions when an application runs.

In summary, our main contributions include the following aspects:

- We propose an approach to change the inherent format of Dalvik instructions and dynamically recover the original instructions, improving the protected capability at instruction level and ensuring the applications work properly.
- We present DynVMDroid, a reinforcement system based on code disorder and dynamic recovery to protect Android applications from instruction analysis attacks.
- Our evaluation demonstrates DynVMDroid is more protective than traditional obfuscating methods, increasing the difficulty of the attack as much as

possible. Besides, we also validate that the scheme is effective in protecting real-world software applications.

2 Related Work

2.1 Classic Code Obfuscation Schemes

There are several common types of code obfuscation including data obfuscation, control flow obfuscation and code virtualization.

Data obfuscation focuses on string encryption and identifier renaming [6–8,13]. As the strings and identifiers in a program usually suggest specific and understandable meaning, scanning engines can take advantage of this feature to understand the code. To avoid this, apk obfuscation tools replace the meaningful strings and identifiers with meaningless ones.

Control flow obfuscation hides the control flow diagram of the original application [7,13,16] to prevent from static analysis by making the application's control flow graph look more complex. A representative approach named control flow flattening (CFF) first appears in Chenxi Wang's thesis [2]. CFF uses a *switch* structure instead of using easily identifiable loops and conditional jumps to transfer the control flow to avoid static and dynamic analysis.

Code virtualization is a typical method of code obfuscation. Zhao et al. [11] presented a scheme that translates the dex bytecode into the common LLVM intermediate representations. Shu et al. [4] proposed a protection scheme named opcode permutation. The scheme is shown in Fig. 1. Each Dalvik instruction includes two parts: opcode and operand. The opcode section specifies the handling procedures to be performed. The mechanism generates a randomly sorted number sequence as virtual opcode and implements a linear mapping between Dalvik opcode and virtual opcode. Zhou et al. [12] proposed a similar approach but extended the length of the virtual opcodes. The idea of code virtualization forces an analyst to move from a familiar instruction set (e.g., arm) to an unfamiliar custom virtual instruction set, hoping to increase the time and effort of reverse engineering.

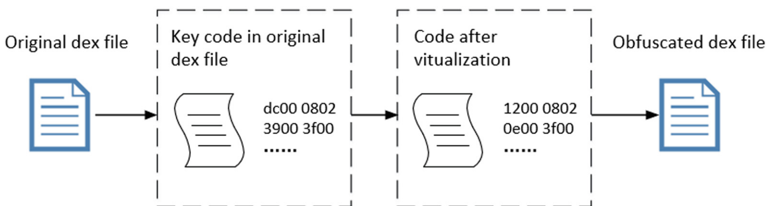

Fig. 1. The classic code virtualization scheme.

2.2 Attack Scheme

With regards to conventional instruction virtualization schemes, the analysts usually perform reverse analysis on the reinforced applications [?] by the following steps. The first step is to locate the entrance to the custom interpreter. As the core module of the virtual execution engine, the interpreter is where the virtual instructions to be executed. Then, it is necessary to find the mapping relationship between each virtual instruction and the handler assigned by the scheduler before execution. Finally, analysts can take advantage of the mapping relationship to recover the original instruction flow of the target code area to obtain the execution logic of the code. For the state-of-art code virtualization scheme mentioned above, when attackers find the opcode mapping table, they can easily restore the original instruction according to the Dalvik instruction structure. In this paper, we call this attack scheme as instruction-intrinsic format analysis.

3 Overview

3.1 Basic Idea

To resist attacks based on instruction-intrinsic format analysis, we break the original structure of instructions by code disorder and dynamically recover the original orders of the opcodes and operands of the instructions. In this way, the attackers cannot recover the original instructions even if they get the opcode mapping table. Detail implementation and algorithms are described in Sect. 4. Figure 2 depicts the architecture of DynVMDroid.

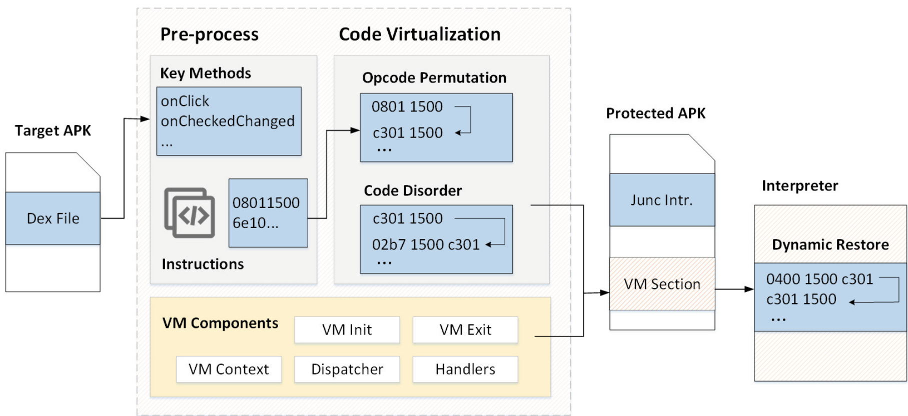

Fig. 2. Overview of the architecture of DynVMDroid. The reinforcement engine is located on the server end that provides reinforcement services for Android applications. The execution part is located on the mobile end where runs the reinforced apk.

DynVMDroid consists of two components, a reinforcement engine and a custom runtime environment. The reinforcement engine mainly disrupts the inherent structural order and extends the length of the original instructions from key methods, converting them into virtual code in Android applications. The custom runtime environment dynamically recovers the virtual instructions to ensure the protected applications work properly. Similar to other VM-based protection scheme, DynVMDroid system should be used to protect the most critical code regions instead of the entire program to minimize run-time overhead.

3.2 Reinforcement Process

The reinforcement engine works as the following steps.

Step 1. Extracting files. The reinforcement system receives an original apk file as input which is unpacked to obtain the dex file and other files such as resource files.

Step 2. Selecting methods. In this step, the dex file is analyzed, while the reinforcement engine selects the key methods to be protected according to the primary rules or specified by the users.

Step 3. Code virtualization. This module first extracts instructions from the method structure. Then, it conduct code permutation and code disorder to generate virtual instructions.

Step 4. Repacking apk. The original instruction stream are cleared from the key methods. Instead, this part are filled with some junk bytecodes. Finally, the processed dex file with other files are repacked in a new apk file.

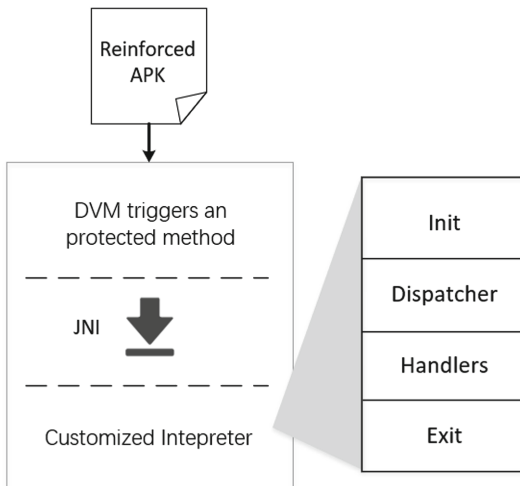

Fig. 3. The execution process

3.3 Execution Process

When the program executes to the protected method, the Dalvik VM calls the corresponding JNI function since the protected method has been changed to the native methods, as shown in Fig. 3. The Dalvik VM transfers execution rights to the custom VM for interpretation and execution. It first initializes the obfuscated codes, decrypting and filling back into the code item field of the method. Then the virtual instructions are executed by the custom interpreter which contains a central loop to schedule, decode, and execute instruction streams. A dispatcher is responsible for identifying each instruction and distributing it to the corresponding handler to execute the virtual instruction. Finally, the custom VM puts back the return value to the Dalvik VM through the JNI interface.

4 Implementation

4.1 Pre-processing Session

Considering the execution efficiency, we select key methods instead of reinforcing all of them. DynVMDroid can protect methods containing sensitive API calls or key algorithms specified by developers. Since we change the original format

Algorithm 1. Metadata Collection Algorithm

Input: *Original_Insns*
Output: *New_Insns*
1: $vm_opcode \leftarrow$ GenRandomVMopcode()
2: $index \leftarrow 0$
3: $extend \leftarrow 0$
4: **for** *original_ins* in *Original_Insns* **do**
5: $\quad rawOpcode \leftarrow$ getOpcode($original_ins$)
6: $\quad pc \leftarrow$ getOffset()
7: $\quad tmpIns \leftarrow$ Replace($original_ins, vm_opcode$)
8: $\quad$ **if** $rawOpcode == OP_GOTO$ **then**
9: $\quad\quad tmpIns \leftarrow$ ExtendToGoto32($tmpIns$)
10: $\quad\quad codeInfo \leftarrow$ InsInfo($pc + extend, index, tmpIns$)
11: $\quad\quad extend \leftarrow extend + 2$
12: $\quad$ **else if** $rawOpcode == OP_GOTO16$ **then**
13: $\quad\quad tmpIns \leftarrow$ ExtendToGoto32($tmpIns$)
14: $\quad\quad codeInfo \leftarrow$ InsInfo($pc + extend, index, tmpIns$)
15: $\quad\quad extend \leftarrow extend + 1$
16: $\quad$ **else**
17: $\quad\quad codeInfo \leftarrow$ InsInfo($pc + extend, index, tmpIns$)
18: $\quad$ **end if**
19: $\quad code_info_dict.push(pc, code_info)$
20: $\quad$ Refresh *index*
21: **end for**
22: **return** *New_Insns*

of the instructions, the Dalvik VM cannot correctly interpret them with a non-standard format. Therefore, we label the key methods as native to generate corresponding JNI functions which will soon be interpreted and executed by our custom interpreter.

4.2 Code Disorder

In this section, we describe some critical techniques of code virtualization at instruction level in DynVMDroid system.

Metadata Collection. We collect some metadata from the original instructions before adopting code disorder transformation. Algorithm 1 is the pseudo-code of the metadata collecting algorithm. First, generate 256 numbers (a random sequence from 0 to 255) required for opcode permutation which is described in Sect. 2. Next, traverse all the instructions. For each instruction, extract the original opcode and record the value of *pc* which indicates the offset of the current instruction in the instruction flow. Then, conduct opcode permutation to get an intermediate instruction before extending *goto* and *goto/16* instructions to *goto/32* instructions in case of data overflow. Afterwards, collect the order of the instruction in one instruction flow as *index*, and the total number of words (each word equals to 2 bytes) extended up to the current instruction as *extend*. Finally, a new structure named *codeInfo* is composed with the metadata collected above. We collect all the *codeInfo* and refresh *index*.

Table 1. Special instructions and payload that need modified and their length and descriptions.

Instruction or payload	Description
if-test	Conditional jump instruction whose offset needs modified
goto	Unconditional jump instruction whose offset needs modified.
goto/16	
try-catch	Exception handling instruction whose start address of the try-item statement and the address of the exception catch handler need modified
packed-switch	Switch-like instruction payload whose address of the corresponding opcode needs fixed respectively.
sparse-switch	
fill-array-data	Arrays padding instruction whose operand indicates the address of arrays

Code Disorder. The most crucial aspect in DynVMDroid reinforcement process is code disorder. In this step, we destructure each instruction in the key

methods, separating one instruction into two parts, which are a word-long starting with opcode and the rest part. Take the instruction *move-object/from16 v1, v21* as an example, whose corresponding bytecode is *0x08011500*. It is decomposed into *0xc301* and *0x1500* after opcode replacement and destructure. We move the two bytes led by the opcode to the end of the current instruction, and finally add two bytes at the beginning of the instruction. The low-order byte indicates the position of the opcode and the high-order byte is a random number. The final form of *0x08011500* is *0x02b71500c301*.

Fix Offset. Code extension brings about variation of bytecode in some instructions which are characterized by the operand part containing offsets. So when dealing with certain instructions, the offset needs fixed. Table 1 shows the instructions and payload characterized by opcodes. Spacial instructions include *if-test*, *goto*, *goto* and *try-catch* whose operands carry offsets needed recalculate and fix. Besides, when the first byte of an instruction is *0x00*, the meaning of the instruction accounts for the second byte whose number might be *0x00*, *0x01*, *0x02* or *0x03*. *0x00* indicates a *nop* instruction. If the second byte is *0x01* or *0x02*, the instruction represents *packed-switch-payload* and *sparse-switch-payload* respectively. Otherwise, the instruction indicates *fill-array-data* when the second byte is *0x03*.

4.3 Custom Interpreter

To ensure the obfuscated methods run properly, we implement a custom interpreter to interpret and execute virtual instructions. Algorithm 2 is the instructions parsing algorithm. For each instruction in virtual instruction set, fix *pc* first which points to the first byte of the current instruction. Next, fetch *index* indicating the position of the opcode. Through the opcode, we can get the operands and determine which handler interprets and execute the instruction. Finally, the scheduler continues to find the next instruction to execute.

Algorithm 2. Instructions Parsing Algorithm

Input: $Virtual_Insns$

1: **for** ins in $Virtual_Insns$ **do**
2: Fix PC
3: $index \leftarrow$ FETCHINDEX
4: $opcode \leftarrow$ FETCH($index$)
5: $operand \leftarrow$ PARSEINS($opcode$)
6: HANDLE($opcode, operand$)
7: **end for**

5 Evaluation

In this paper, we evaluate the reinforcement function in four aspects of functionality, compatibility, security, performance. They are defined as the following concepts.

- Functionality: Whether the function of the application changes after reinforcement.
- Compatibility: Whether the reinforced application can run stably on the Android OS.
- Security: The ability to withstand attacks from reverse engineering.
- Performance: The execution overhead of the reinforced application.

We evaluate DynVMDroid and observe what extent it meets these four criteria.

5.1 Environment Setting

We applied DynVMDroid to 10 different applications without any types of protection in the real world. We were limited to 10 applications because we have to manually verify their correctness and efficiency respectively. Due to the complexity of testing and the limitations of experimental equipments, all cost-related experiments were conducted on Android 10 and Android 12 platforms according to statistics.

5.2 Compatibility and Functional Analysis

First, we installed the applications protected by DynVMDroid on experimental equipments to test the main process, main functions and main interface, observing whether there were any abnormalities on runtime. During the testing process, we did not find that the programs crashed or had any obvious freezes, and the function of the application did not change after the reinforcement.

5.3 Security Analysis

Manual Attack. We manually analyze a method protected by DynVMDroid from the perspective of an analyst. First, we obfuscate a function *onClick* with a statement *Toast* in it and packed it into an apk. Then, we employ reverse engineering on the reinforced apk file with the following steps.

Step 1. Start dynamic debugging via IDA Pro. Since the protected function is extracted, it cannot be obtained only by static analysis. Therefore, we adopt dynamic analysis via the state-of-the-art disassembler IDA pro [5] on the app and find the app using VMP technique to protect onClick function.

Step 2. Find the start address of the protected function. We need to add a breakporint at JNIMethodStart in order to get the start address of the protected JNI function from memory.

Step 3. Recover information of the protected function from memory. In this step, we suppose the analyst find the mapping between the original opcode and the virtual opcode. After recovering the opcodes, we script to dump the corresponding code from the memory to recover the function.

Step 4. Analyze the recovered file. We use IDA pro to disassembly the recovered dex file. As shown in Fig. 4, we can find the comparison from the pseudo-code output result. The left part of Fig. 4 shows the disassembly result of

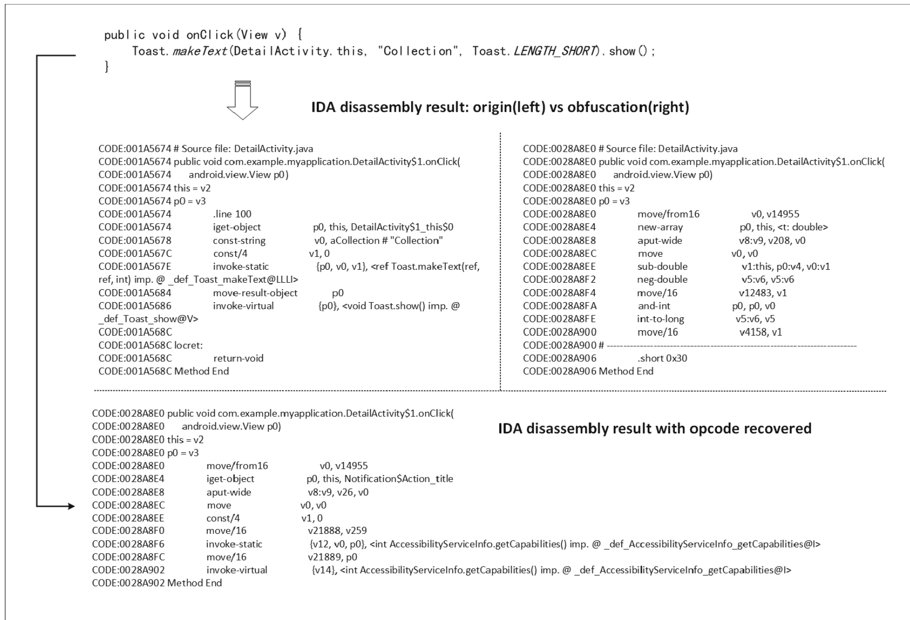

Fig. 4. Comparison of original program and its static decompilation results for IDA pro and disassembly results for opcode recovered

original function. In contrast, the obfuscated instructions are completely different from the original instructions as shown on the right part. The instructions on the bottom of Fig. 4 indicate the results for recovering the opcodes, we still cannot get any information about the original instruction from it.

Deobfuscation Tools. We employ 4 representative deobfuscating tools on DynVMDroid and other obfuscating schemes. The tools include Simplify [3], Deoptfuscator [9], TIRO [10], dexMonitor [17]. Table 2 shows the result of the test. Simplify executes the code to be decompiled through simulating Dalvik VM, learns its function and then simplifies the decompiled code into an understandable form. It is commonly used to parse strings such as key function names and regular

Table 2. Test results with deobfuscation tools

	Identifier Renaming	control flow obfuscation	classic code virtualization	code disorder
Simplify	✓	✗	✗	✗
Deoptfuscator	✗	✓	✗	✗
TIRO	✓	✓	✗	✗
dexMonitor	✓	✓	✓	✗

expressions, but not effective for control flow obfuscation and code virtualization. Deoptfuscator is designed to converted obfuscated applications with the control flow obfuscation mechanism, especially performed by dexGuard on open source Android applications. However, it is not available on the other obfuscation schemes. TIRO targets locations where obfuscation may occur, applies instrumentation to monitor for obfuscation, collects run-time information and then produce deobfuscated code. However, TIRO may not be able to extract all targeted paths and constraints due to static imprecision and complex path constraints in the code. Dexmonitor places hooks where a Dalvik instruction is about to be executed so that a view of the disclosed code and data can be generated when the program counter reaches this point. However, the idea can not be applied on code virtualization because the disclosed code is still virtualized in Dalvik VM.

5.4 Cost

This study provides a comprehensive evaluation of the performance of the reinforcement system, which is mainly tested from three aspects. First, we compares

Table 3. The data set and its experimental results for runtime performance

System	Application	CPU(%)		Memory(MB)		
		Before	After	Before	After	Increase(%)
Android 10	XAPK Install	6.72	4.68	80.78	88.66	9.75
	Auto Click	4.07	6.95	81.72	94.63	15.80
	Skin Tool	18.64	23.19	219.54	195.20	−11.07
	MirrorPlus	15.24	25.20	217.29	219.11	0.84
	RandoChat	9.25	15.78	121.46	127.92	5.32
	Calendar	20.39	15.31	163.13	156.44	−4.10
	Calculator	3.52	3.53	26.01	27.76	6.73
	Diary	3.68	5.68	120.20	141.46	17.69
	Music Player	15.57	16.33	148.72	158.85	6.81
	File Transfer	2.57	4.48	80.42	82.76	2.90
Android 12	XAPK Install	–	–	–	–	–
	Auto Click	7.18	11.15	207.59	211.40	1.84
	Skin Tool	7.51	14.02	242.72	291.58	20.13
	MirrorPlus	15.53	14.06	351.49	339.26	–3.48
	RandoChat	13.12	18.52	311.62	324.81	4.23
	Calendar	38.70	25.90	312.71	403.16	28.92
	Calculator	9.29	8.93	140.07	137.12	−2.11
	Diary	10.63	11.67	240.75	271.92	12.95
	Music Player	26.84	28.78	326.07	389.44	19.43
	File Transfer	–	–	–	–	–

the changes in APK file size of protected applications before and after hardening. Second, we measure the increment of CPU utilization and memory usage when the application is running before and after hardening, so as to evaluate the impact of reinforcement on the runtime performance of the application. Through these tests, the performance of the hardened system can be fully evaluated.

Increment in Size of APKs. The size of the APK file can have an impact on the download, installation, and update speed of the application, as larger APK files take longer to download and install. Larger APK files may cause the device to run out of storage space, making it impossible to install applications or prevent other applications from functioning properly. Therefore, when developing and optimizing applications, we need to consider the size of the APK file and minimize unnecessary resources and code to improve the performance and user experience of the application.

Figure 5 compares the changes in APK file size of protected applications before and after hardening. We can see from Fig. 5 that although some files (such as encrypted virtual instructions and custom interpreter files) are newly generated during the hardening process, the increase in the size of the APK file is still within the normal range and will not have a great impact on user's installation and use. In protected applications, there is a certain correlation between the increment of APK file size and the complexity and function increment of the application before and after hardening. Since each instruction in the protected function body is expanded, the longer the sequence, the larger the increment of the APK file after hardening.

Runtime Overhead. In this experiment, we randomly simulated clicks for each protected application through the Monkey [14] tool, and obtained the CPU resource and memory resource usage at the application runtime with the Mobileperf [15] tool. Finally, we recorded and analyzed the results. Table 3 shows

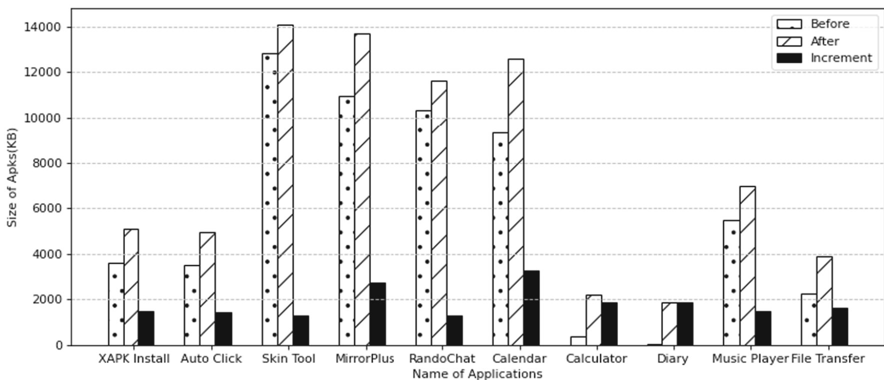

Fig. 5. Comparison of APK file size before and after application obfuscation

the data set and the results for runtime performance, noted that the two original applications of XAPK Install and File Transfer are not competitive on Android 12 platform so we did not record the results.

From Table 3, we can see that the average CPU occupancy of the protected applications tends to be larger than the original applications, mainly derived from the decryption and interactions with the custom interpreter. Nonetheless, the volume of changes are within an acceptable range. After reinforcement, the overall memory usage of hardened applications is slightly higher than that of original applications. This is because when the application runs to a protected method, decryption operations will be performed. At this time, the virtual instruction sequence will be decrypted and loaded into the memory, resulting in an increase in memory usage. However, the runtime memory usage of hardened applications is still within the normal range, and has not caused a major impact on system operation.

6 Discussion

In this section, we discuss the limitations of DynVMDroid, possible solutions and future work. First, analysts can run the obfuscated method by dynamic debugging and observe the register variables to infer the rules of instruction transformation. Although this manual analysis process is very time-consuming and challenging, this intrinsic makes it possible to recover the program bytecode. We can defend this attack by adding anti-debugging mechanism. Besides, the VM may reduce the efficiency of the program's operation not to mention that we employ code virtualization on the application. If the application is heavy and rich in functionality, we can only reinforce a few of the methods and avoid the functions with high time complexity. In future work, we will simplify the interpretation process of instructions to enhance the performance of program execution. In addition, we might as well improve the prototype and compare it with enhanced reinforcement system.

7 Conclusion

This paper proposes an Android application reinforcement system DynVMDroid based on code disorder and dynamic recovery, which further improves the security of the application at the instruction level. We have described the architecture of DynVMDroid and its implementation to overcome the challenges while transforming the structure of instructions. The system requires two assemblies including the reinforcement engine which disorder the instructions in key methods and the runtime environment responsible for dynamic code recovery and execution. We evaluate DynVMDroid from four aspects, including compatibility, functionality, security and performance on 10 applications. The experimental results show that the target programs protected by DynVMDroid are effectively resist manual attacks and the state-of-the-art deobfuscation tools. Besides, it achieves acceptable range of code size, cpu occupancy and memory usage.

References

1. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. In: 2001 International Conference on Dependable Systems and Networks, pp. 193-202 (2001). https://doi.org/10.1109/DSN.2001.9
2. Simplify. https://github.com/CalebFenton/simplify. Accessed 11 Apr 2022
3. Shu, J., Li, J., Zhang, Y., Gu, D.: Android app protection via interpretation obfuscation. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, pp. 63-68 (2014). https://doi.org/10.1109/DASC.2014.20
4. Eagle, C.: The IDA Pro Book. No Starch Press, San Francisco (2011)
5. Tang, Z., Chen, X., Fang, D., Chen, F.: Research on java software protection with the obfuscation in identifier renaming. In: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), pp. 1067-1071 (2009)
6. dexGuard. https://www.guardsquare.com/dexguard. Accessed 11 Apr 2022
7. Stringer JAVA obfuscator. https://jfxstore.com/stringer. Accessed 11 Apr 2022
8. Deoptfuscator. https://github.com/Gyoonus/deoptfuscator. Accessed 11 Apr 2022
9. Wong, M.Y., Lie, D.: Tackling runtime-based obfuscation in Android with tiro. In: Proceedings of the 27th USENIX Conference on Security Symposium, pp. 1247–1262. SEC'18, USENIX Association, U (2018)
10. Zhao, Y., et al.: Compile-time code virtualization for Android applications. Comput. Secur. **94**, 101821 (2020). https://doi.org/10.1016/j.cose.2020.101821
11. Zhou, W., Wang, Z., Zhou, Y., Jiang, X.: Divilar: Diversifying intermediate language for anti-repackaging on Android platform. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, pp. 199–210. CODASPY '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2557547.2557558
12. Aonzo, S., Georgiu, G.C., Verderame, L., Merlo, A.: Obfuscapk: an open-source black-box obfuscation tool for Android apps. SoftwareX **11**, 100403 (2020). https://doi.org/10.1016/j.softx.2020.100403
13. UI/Application exerciser monkey. https://developer.Android.com/studio/test/monkey. Accessed 15 Mar 2020
14. Mobileperf. https://gitee.com/sanmejie/mobileperf. Accessed 11 Apr 2022
15. Yang, X., Zhang, L., Ma, C., Liu, Z., Peng, P.: Android control flow obfuscation based on dynamic entry points modification. In: 2019 22nd International Conference on Control Systems and Computer Science (CSCS), pp. 296–303 (2019). https://doi.org/10.1109/CSCS.20
16. Cho, H., Yi, J.H., Ahn, G.J.: DexMonitor: dynamically analyzing and monitoring obfuscated Android applications. IEEE Access **6**, 71229–71240 (2018). https://doi.org/10.1109/ACCESS.2018
17. Xue, L., et al.: Parema: an unpacking framework for demystifying VM-based android packers, pp. 152–164. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460319