IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 10, 2019, accepted May 23, 2019, date of publication June 6, 2019, date of current version August 30, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2921417

Exploiting Binary-Level Code Virtualization to Protect Android Applications Against App Repackaging

ZHONGKAI HE[1,2], GUIXIN YE[1,2], LU YUAN[1], ZHANYONG TANG[1,2], XIAOFENG WANG[1], JIE REN[3], WEI WANG[1,2], JIANFENG YANG[1], DINGYI FANG[1,2], AND ZHENG WANG[4,5]

[1]School of Computer Science and Technology, Northwest University, Xi'an 710069, China
[2]Shaanxi International Joint Research Centre for the Battery-Free Internet of Things, Xi'an 710127, China
[3]Shaanxi Normal University, Xi'an 710062, China
[4]Lancaster University, Lancaster LA1 4YW, U.K.
[5]Xi'an University of Posts and Telecommunications, Xi'an 710121, China

Corresponding authors: Zhanyong Tang (zytang@nwu.edu.cn) and Xiaofeng Wang (xfwang@nwu.edu.cn)

This work was supported in part by the NSFC under Grant 61672427 and Grant 61672428, in part by the International Cooperation Project of Shaanxi Province under Grant 2019KW-009 and Grant 2017KW-008, in part by the Key Research and Development Project of Shaanxi Province under Grant 2017GY-191, in part by the Shaanxi Science and Technology Innovation Team Support Project under Grant 2018TD-O26, in part by the China Scholarship Council under Grant 201806970007, and in part by the Ant Financial through the Ant Financial Science Funds for Security Research.

ABSTRACT Application repackaging is a severe problem for Android systems. Many Android malware programs pass the mobile platform fundamental security barriers through repackaging other legitimate apps. Most of the existing anti-repackaging schemes only work at the Android `DEX` bytecode level, but not for the shared object files consisting of native ARM-based machine instructions. Lacking the protection at the native machine code level opens a door for attackers to launch repackaging attacks on the shared libraries that are commonly used on Android apps. This paper presents CodeCloak, a novel anti-repackaging system to protect Android apps at the native code level. CodeCloak employs binary-level code virtualization techniques to protect the target application. At the native machine code level, it uses a newly designed stack-based virtualization structure to obfuscate and protect critical algorithm implementations that have been compiled into native instructions. It leverages multiple dynamic code protection schemes to increase the diversity of the program behavior at runtime, aiming to increase the difficulties for performing code reverse engineering. We evaluate CodeCloak under typical app repackaging scenarios. Experimental results show that CodeCloak can effectively protect apps against repackaging attacks at the cost of minimum overhead.

INDEX TERMS Android code protection, code obfuscation, app repackaging, code virtulization.

I. INTRODUCTION

Application repackaging is a prevalent and severe threat to the Android ecosystem. With the help of dynamic profiling and reverse engineering tools, an attacker can unpack an app, replace and insert code to, e.g., remove advertisements, steal privacy information, or make purchases without the user's authorization [1]. A prior study shows that over 80% of the malware samples were implemented through repacking legitimate apps [2]. Therefore, there is a critical need to protect Android apps from repackaging attacks.

The associate editor coordinating the review of this article and approving it for publication was Bora Onat.

Code obfuscation is a viable means to protect applications against reverse engineering and repackaging [3]. By creating code that preserves the intention and semantics of the original code but is challenging to understand, code obfuscation increases the time and efforts for performing code reverse engineering. There is considerable work in applying code obfuscation to protect Android applications against repackaging. In Figure 1, we summarize some of the most relevant work. Many of the previous approaches target at the Android `DEX` bytecode level. Proguard [4] and DexGuard [5] are two representative work, which, however, cannot effectively protect the obfuscated code if the entry point (such as the `memcpy` method) is found using tools like

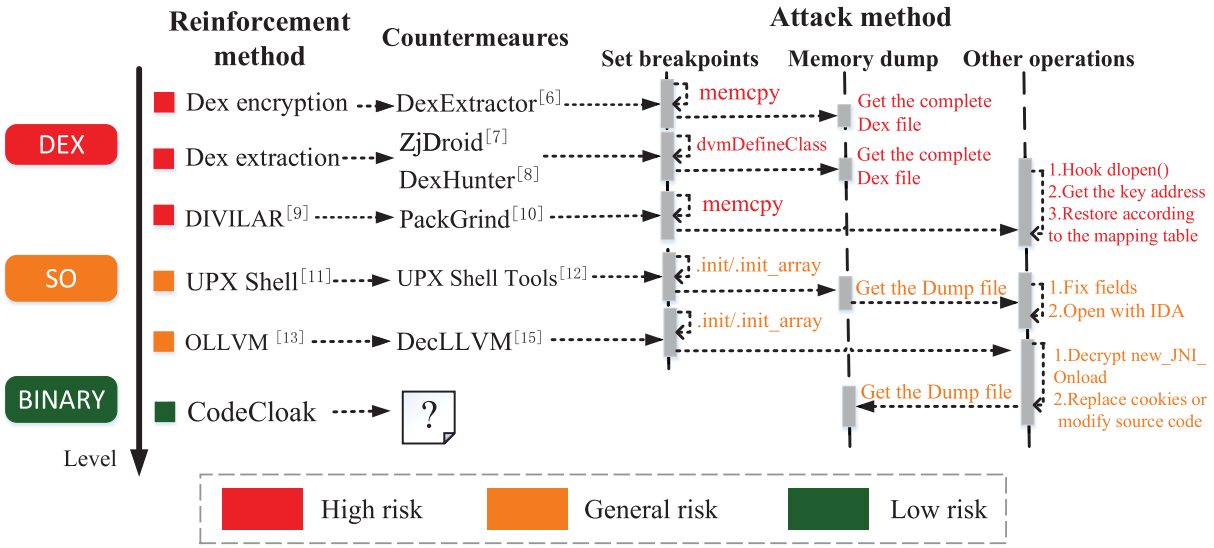


FIGURE 1. Summary of prior code protection schemes their corresponding attacks for Android apps. Here, different color blocks represent levels of different risks. A red color indicates high risk while the green suggests the risk to be low. Most protection systems do not target the binary level.

DexExtractor [6]. Other implementations overwrite the way a DEX file is loaded by changing the Android DEX class loader, aiming to increase the difficulties for observing standard function calls. However, an attacker can still bypass such a defense by debugging the native commands through tools like ZjDroid [7] or DexHunter [8] to observe standard library calls. DIVILAR [9] was able to protect apps against function call observations, but it is proven to be vulnerable under more advanced tools like PackGrind [10] that can decrypt and reconstruct the mapping between code semantics and function calls.

In addition to DEX files, there are many Android apps built upon shared libraries which were firstly written in high-level languages like C and C++ and then compiled into native machine instructions. These shared libraries often implemented the frequently used core algorithms. Therefore, there is a need to protect share object (SO) files against code reverse engineering and app repackaging. However, existing SO protection schemes often adopt a simple but less effective code obfuscation or encryption strategy. They do not provide sufficient protection against sophisticated code reverse engineering attacks. For example, UPX shelling [11] is one of such protection methods, but an attacker can use the UPX Shell tools [12] to launch the attack as shown in Figure 1. OLLVM confusion [13] is compiler-based code obfuscation performed at the source code level, but it is proven to be vulnerable under new anti-obfuscation methods [14], [15].

This paper aims to propose a better code obfuscation approach for SO files. Our work targets applications compiled for the ARM instruction set, a de-facto Android hardware architecture. As a departure from prior work, our code obfuscation scheme, namely CodeCloak, works at the binary level. It employs a stack-based virtualization scheme to protect the logic of algorithms and protocols implemented in SO files. At the native machine code level, it uses a novel stack-based virtualization structure to protect native ARM instructions. To enhance the security strength, we adopt multiple virtual protection schemes, where a scheme is dynamically chosen at runtime. Furthermore, our implementation is fully compatible with existing protection schemes for DEX. As a result, CodeCloak closes the gap between DEX and SO file protection.

We evaluate CodeCloak under typical app repackaging settings. Our evaluation results show that CodeCloak can effectively protect apps from repackaging attacks, and it achieves this at the cost of minimum overhead. One of the key contributions of this paper is a novel approach for protecting native share object files against app repackaging on Android systems. The other contribution is the first approach for binary-level code virtualization for ARM instructions, and it can be applied to many embedded systems that are powered by ARM processor architectures.

II. BACKGROUND

A. VM-BASED ANDROID APP PROTECTION SHCEME

The VM protection process consists of the following steps. We first decompile the binary SO file and extract the key ARM instructions according to the pre-set tags. Then, the extracted ARM instructions are mapped to virtual instructions which are still turning equivalent. Next, the virtual instructions are encoded into the SO file in a binary form utilizing the custom encoding rules. Finally, the combined custom interpreter is inserted in a binary SO file. By using these strategies over the binary SO file as it is shown above, the VM-based protection scheme will effectively increase the attack cost of the attacker.

To illustrate operations of the scheme, we take DIVILAR [9] as an example. DIVILAR is a VM-based protection method, and it converts the real instructions into a virtual instruction set and adds a hook mechanism to restore

```
...
0x112E  CMP   R7,#0xD1
0x1130  ASRS  R6,R6,#4
0x1132  ASRS  R5,R5,#7
0x1134  LSRS  R4,R2,#0XD
0x1136  LSRS  R3,R7,#0X15
0x1138  LDRH  R3,[R5,R5]
0x113A  BVC   _ZN7_JNIEnvGetStringUTF
0x113C  B     loc_113E
...
```

(a)StirngFromJNI after UPX shelling.

```
...
0x112A  LDR   R2,[SP,#0x50+var_34]
0x112C  STR   R3,[SP,#0x50+var_38]
0x112E  STR   R4,[SP,#0x50+var_3C]
0x1130  STR   R5,[SP,#0x50+var_40]
0x1132  BL    _ZN7_JNIEnvGetStringUTF
0x1136  STR   R0,[SP,#0x50+var_1C]
0x1138  CMP   R0,#0
0x113A  BNE   loc_1144
0x113C  B     loc_113E
...
```

(b)The repaired StirngFromJNI method.

FIGURE 2. Comparison before and after reverse peeling. Red-marked instructions represent encrypted, unrecognized instructions.

and interpret the virtual instructions at runtime. Although DIVILAR provides protection only in the level of DEX file, it also proves that the VM-based security method is effective against common countermeasures, including static analysis, dynamic analysis, and specific analysis for virtual machines.

Before showing the SO protection crack example, we must point out that the most significant difference between the scheme of CodeCloak and DIVILAR is the protection objects are completely inconsistent. The object protected by DIVILAR is the DEX file, while our system protects the lower level SO file. Due to VM protection running in DEX file, DIVILAR must use a hook mechanism to communicate within components. However, it is a design defect that an attacker can utilize this mechanism to obtain information between instructions during the translation process. CodeCloak works in the level of SO file and avoids problems caused by the instruction restoration process when the program is interpreted.

B. SO PROTECTION CRACK EXAMPLE

As mentioned above in Figure 1, we have understood that SO file protection scheme is more effective than that of DEX file, but we would like to know the existing method could take protection effective or not? To ameliorate our doubts, as described next, we will manually attack several common protection methods in SO files. Here we use the interactive disassembler IDA Pro [16] to dynamically debug and analyze SO files.

At present, the particular protection scheme of SO files mainly employ encryption, as we all know, the deformation of UPX shell [11] is one of the most commonly used forms above in existing app reinforcement manufacturers. Here we take the SO file protected by Ijiami [17] as the attacking object. Figure 2 (a) shows partial instructions of the protected function `stringFromJNI`.

As we can see, for the entire instructions are encrypted and erroneous, it is very confusing for the adversary to understand the semantics of the code section. However, as a skillful cracker, he/she could first analyze the loading mechanism of the SO file, then create a dump point before the `.init/.init_array` in the memory. Contrary to static analysis attacks, this method could fix the corresponding load, dynamic, and section fields. After this series of operations, Figure 2 (b) shows the repaired `StringFromJNI` method opened with IDA Pro.

OLLVM obfuscation is accomplished by hiding the real control flow of the application. We choose the SO file protected by Tencent Legu [18] as the attacking target where the `JNI_Onload` method in the SO file is confused.

To implement the attack, the adversary directly set the breakpoint at `.init/.init_array`, he/she can debug and decrypt the `JNI_Onload` method function. Using this method, we can obtain `new_JNI_Onload` that is a new "load" function decrypted from `JNI_Onload`. Alternatively, the cracker can use the replacement cookie (Dalvik mode) or modify the source code (ART mode) to attack, and finally repair the dumped file while getting the decrypted info. Hence, in this case, we can draw a clear conclusion that the obfuscation based on OLLVM is more difficult to be debugged than the UPX shell, but experienced attackers can still bypass these protection mechanisms through dynamic debugging.

These two crack cases show that it is urgent to persevere a protection method to prevent both static analysis and dynamic debugging. Attack experiments of this paper in section IV show that CodeCloak can prevent a cracker from conducting above two kinds of breakdown, and even prevent special attacks on virtual machines.

C. THE ATTACK MODEL

The existing research [19] has illustrated the reverse steps of the VM-protected program. Here we summarize as follows:

`Step 1`: to find the confused entry point address of the VM interpreter;

`Step 2`: to find the address of the dispatcher and restore the handlers executed at runtime, record the handler addresses. The cracker will discover the mapping relationship between the virtual and the real instructions;

`Step 3`: using the knowledge obtained from the first two steps to recover the logic of the target code region. These steps are the basic operations for an attacker to launch an attack.

Our attack model assumes that attackers have practical experience in software reverse engineering. We assume that attackers can use our protection program to protect any Android applications multiple times in a specific application environment. We also assume that attackers can debug, track, and modify binary SO files in memory through analysis tools such as IDA Pro [16] and Valgrind [20]. In a word, the purpose of the attackers is to implement the attacks through the analysis of VM's working mechanism and the transformation logic between instructions. Contrary to the purpose of the attackers, our goal is to protect VM's working principles and mapping schemes between instructions from being discovered by using as reliable protections as possible.

III. DESIGN OF CODECLOAK

A. OVERVIEW OF OUR APPROACH

To address the problems of repackaging attacks, we propose a system which we coin CodeCloak, a native ARM instruction virtualized system for Android apps. The purpose of

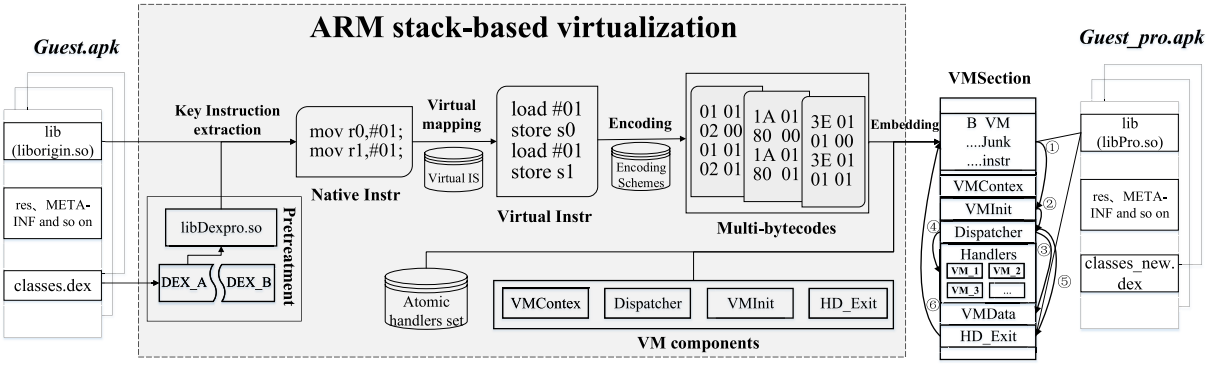


FIGURE 3. Overview of CodeCloak. The middle area highlighted by gray rectangle shows ARM stack-based virtualization, including pre-processing, key instruction extraction, multiple virtualizations, building and embedding the interpreter VMSection. The VMSection is executed as following steps: ① Jump into the virtual machine; ② Initialize the VM, enter the Dispatcher; ③ Read the virtual instruction bytecodes; ④ Dispatch handlers to process bytecodes; ⑤ Exit the virtual machine; ⑥ Go to the subsequent instructions to continue execution.

CodeCloak is to provide protection for Android applications at the lower and deeper binary code levels.

Figure 3 depicts the overall system architecture of CodeCloak. It takes the APK as input and binds the virtualized binary file to the compiled APK as output. As mentioned above, CodeCloak focuses on protection policies of Android native `SO` files, including the original `SO` file and the shell file after `DEX` protection such as DIVILAR [9]. We can divide CodeCloak's protection process into multiple virtualization modules and construct custom interpreter engine modules. The former is to convert the original ARM instructions into the virtualized instructions by selecting a mapping rule from multiple sets of the custom mapping conversion rules, and the latter constructs the interpreter which interprets and executes virtualized instructions while the program is running.

B. ARM VIRTUAL-MACHINE-BASED PROTECTION

Unlike the Java virtual machine, CodeCloak is a stack-based virtualization protection scheme for ARM instructions. This protection scheme can be implemented through the following two phases: (1) utilizing multiple virtualization module to perform code translation, and (2) creating a custom interpreter engine.

In the first phase, CodeCloak rewrites the instructions to a new form, which inserts mapping tables, atomic handles, and then packages `.apk` static file. To be specific, the system first disassembles the key HEX and get the ARM instructions from the pre-set label code segments or the start and end addresses of `Guest.apk`. After applying these mapping rules, the system could transform the native instructions into the virtualized formats one by one. Concerning these two types are Turing-equivalent; in general, we usually employ multiple custom instructions to simulate a single native instruction. As Figure 3 shows, we can quickly draw the conclusion that the mapping rules are very important. To avoid being cracked, multiple sets of mapping rules and corresponding interpretation handlers are designed, which instead of a single mapped ARM virtual machine. Using the method above, every original protected instruction can be converted to virtual instructions of different code sets each time.

In the second stage, CodeCloak implements a global abstract interpretation engine `VMSection` in a protected `SO` file. `VMSection` can be considered as a code pump, which is accomplished by simulating various kinds of schedule functions on real CPU. `VMSection` contains six components: the converted custom virtual instructions `VMData`, the initialize program `VMInit`, the register environment `VMContext`, the virtual machine's scheduler `Dispatcher`, the exit program `HD_Exit`, and the corresponding operations `Handlers`. `VMSection` embeds `SO` files in binary form. Worthy of note is that the beginning of the code region will be filled with `B VM` and junk instructions to erase the traces of the original execution. Finally, CodeCloak outputs a brand new virtualized `SO` file.

Thus, in summary, if an attacker wants to crack this kind of virtualization mechanism, he/she must grasp all information completely through the internal working principles of the custom interpreter and the functions of each virtual instruction. Meanwhile, he/she has to restore the original functions of the running virtual instructions, which is a cumbersome and error-prone process that can easily trigger the domino effect.

C. TIME DIVERSITY

Time diversity is one of CodeCloak's design goals, and which devotes the system to build multiple sets of bytecode instruction and handlers during the protection code executing. Once the system generates multiple instruction sets, the Dispatcher randomly selects one type of those to perform every round.

In contrast, the classic VM protection scheme [21] uses the dispatcher to acquire and parse the compiled bytecode instructions during execution. We must point out that the fetched instructions set are unaltered; in other words, the operation code of bytecode instructions is delivered over to a fixed handler. Since there is a one-to-one correspondence between custom instructions and interpreters, the attacker can

TABLE 1. An example of NI and corresponding structure.

NI[1]	VI[2]	VMData		
		Rule1	Rule 2	Rule 3
	load_reg r0	0x1B,0x00,	0x4D,0x00,	0x2F,0x00,
add r0,r0,r1	load_reg r1	0x1B,0x01,	0x4D,0x01,	0x2F,0x01,
	vadd	0x2D,	0x0F,	0x1C,
	store r0	0x02,0x00	0x2E,0x00	0x3B,0x00

[1] In this table, NI indicates the native ARM instructions.
[2] In this table, VI indicates the virtual instructions.

quickly get the mapping rules of the real instructions and the virtual instructions.

We address all of these issues in prior VM protection scheme and provide empirical evidence of our ability to solve problems beyond the reach of previous methods. CodeCloak establishes mapping tables between multiple sets of custom instructions and interpreters at the same time. We automatically generate many alternative implementations for each handler. These implementations can generate equivalent results for the original input instructions. Finally, the protected program will randomly select a mapping table to execute with the Dispatcher and handlers of the interpreter.

As a simple example consider the following add instruction, in this case, this native instruction mainly completes the add operation of the `r0` register and `r1` register, and the sum will be restored in the `r0` register. These operations will be divided into `load_reg`, `vadd`, and `store` virtual instructions after being virtualized by CodeCloak. The `load_reg` instruction is used to push the operand to the stack, and the `store` instruction is used to pop the stack and store the result in the `VMContext`. According to different encoding rules, the virtual instructions are randomly encoded into one of the three bytecode formats as shown in Table 1. At the same time, we generate corresponding sets of handlers for each VM which are functionally equivalent in nature. However, the same bytecodes have different meanings in different VMs. We would like to add that the number of VMs is flexible according to the runtime performance overhead and protection strength of the app. A protected application with diversity and uncertainty can easily invalidate the knowledge gained by an adversary from previous reverse attacks.

D. AN EXAMPLE

1) PROTECTION PROCESS

To illustrate how CodeCloak protects the APK, we have selected a partial code snippet of `2048.apk` [22] to explain the protection process. The steps of CodeCloak's protection process are described as follows.

`Step 1`: As shown in Figure 4, we first unzip the application installation package to get the binary `SO` file, then disassemble it to select the key code segment. To simplify the problem, we assume that the code fragment has a starting offset address `0x17A8` and an ending offset address `0x19B8`.

```
.text:000017A8   EXPORT Java_com_implementist_game_game2048_GameView_swipeRight
.text:000017A8 Java_com_implementist_game_game2048_GameView_swipeRight
.text:000017A8          STMFD           SP!, {R4-R11,LR}
.text:000017AC          ADD             R11, SP, #0x1C
.text:000017B0          SUB             SP, SP, #0x64
.....
.....
.text:000019B0          STR             R0, [SP,#0x80+var_74]
.text:000019B4          ADD             R0, R10, #0x10
.text:000019B8          STR             R0, [SP,#0x80+var_78]
.text:000019BC          B               loc_19D4
.....
```

FIGURE 4. Decompile key code segments to be protected.

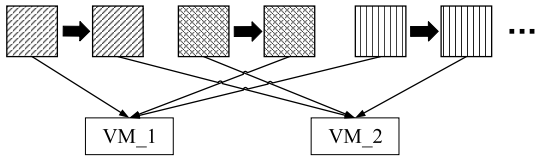


FIGURE 5. Schematic diagram of handlers embedded in VMs under custom configuration and the same striped boxes represent the functionally equivalent handler.

`Step 2`: Next, the ARM instructions of the segments being protected are virtualized one by one. As mentioned above, using encoding rules, the designed scheme will randomly select and embed the virtual instructions from the multiple sets into the `VMData` in binary bytecode format.

`Step 3`: Generate the corresponding VM according to the custom configuration choice. As shown in Figure 5, if the configuration choice is `VM_2`, it will automatically generate a functionally equivalent implementation for each handler. In fact, for each running, every VM has different handles execution sequences, and the mapping tables between the virtual instructions and the handlers are also highly dynamic. Namely, the control flow undergoes continuous changes.

`Step 4`: In order to cover up the real jumping entry point of the virtual machine, the obfuscation is implemented with the insertion of garbage instructions at the associated particular location in the binary.

`Step 5`: The system embeds a new code fragment `VMSection` into the `SO` file. `VMSection` is composed of `VMInit`, `VMData`, several VM structures just established, etc.

`Step 6`: In the last step, the system repackages the new app. The function of the newly generated version `new_2048.apk` is equivalent to the pre-protection one.

2) RUNTIME EXECUTION FLOW

To illustrate how a protected app executes, we use the newly generated `new_2048.apk` as an example. In what follows, we describe the technical details of our case.

`Step 1`: When `new_2048.apk` executes to the protected code segment, the `B VM` instruction is executed first. As described in Figure 3, the protected application jumps to `VMInit` component to initialize the virtual machine. Specifically, for the correct restoration after completing executions of protected instructions, the values of the register in the current runtime environment are saved to `VMContext` for simulating the behavior of the real CPU register.

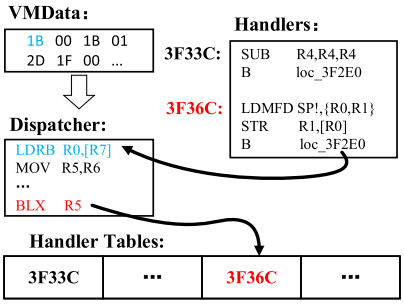


FIGURE 6. The scheduling process of the dispatcher.

`Step 2`: After the `VMContext` is initialized, the Dispatcher executes the instruction bytecodes in the `VMData`. For the bytecodes have been virtualized, CodeCloak needs to choose a corresponding set of VMs to perform in accordance with the coding rules of the virtual instructions. As can be seen in Figure 6, here we choose the first set of rules (`VM_1`) to explain. Once a decision has been made in the choice of VM, we will further select the corresponding handler to execute based on the parsed opcode. Let us take the add instruction parsing process in Figure 6 as an example, the `R7` register points to the address of `VMData`. After parsing the content of the `R7` register, the program starts executing the handler with the offset address `0x3F36C`.

`Step 3`: The interpreter of VM starts to select and execute the first handler. For one original instruction is mapped into semantic equivalents in several virtual ones, the system will execute multiple handlers. As shown in Figure 6, when an epoch successfully completes execution, the program jumps to the Dispatcher's entrance address `0x3F2E0`.

`Step 4`: Inside the loop, the program executes the byte in `VMData`. Once all the bytecodes are fetched out, the control jumps to take `Step 5` otherwise repeats `Step 2`.

`Step 5`: After interpreting and executing all `VMData` bytecodes, the program jumps to `HD_Exit`, the function of this point is to restore the latest `VMContext` values to the real registers.

`Step 6`: The program jumps back to the end point address of the protected code segment and continues performing outside instructions of the virtual machine.

IV. EVALUATION

CodeCloak provides effective protection on the binary level for Android apps against repacking. In this section, we comprehensively evaluate the effectiveness of CodeCloak by analyzing the security and performance of applications protected by CodeCloak.

A. GENERAL ANALYSIS

In this section, we analyze the effectiveness of CodeCloak from two aspects. On the one hand, CodeCloak uses atomic operations to interpret the core instructions of the app to be protected. This is difficult for prior attacking methods to repack the protected app because atomic operations carry little semantic information. On the other hand, we analyze the app protected by CodeCloak and six popular commercial protection tools. Table 2 shows the results of the analysis, which shows that CodeCloak is the only one protection approach to use the multi-virtualization technique to protect `SO` files.

TABLE 2. Comparison of mainstream reinforcement.

	Qihoo	Ijiami	Ali	NetEase	Baidu	Tencent	CodeCloak
SO Virtualization	×	√	×	√	×	×	√
SO Diversity Virtualization	×	×	×	×	×	×	√

```
.nisl:00006384          STMFD SP!, {R3} .nisl:00006388
B loc_62E4 .nisl:0000638C ;---------------------------- .nisl:0000638C
LDMFD SP!, {R0} .nisl:00006390          ADD SP, SP, R0 .nisl:00006394
STR SP, [SP] .nisl:00006398          B loc_62E4 .nisl:0000639C
;----------------------------
```

FIGURE 7. Disassembly of the new `.nisl` code segments after virtualization.

TABLE 3. Dynamic shelling tools and features.

Tool name	Open source	Require root	Principle
DrizzleDumper	√	√	Characteristics of the `DEX` file search
ZjDroid[1]	√	√	Shelling based on Hook injection
DexExtractor	√	×	Modify the system method based on hook
DexHunter	√	×	Proactive load and initialize the `DEX` file
PackerGrind	×	×	Multi-layer unpacking detection
DroidUnpack	×	×	Multi-layer unpacking detection

[1] ZjDroid only opens the code of the Java layer, but the code of the native layer is not open yet.

B. STATIC ANALYSIS

Before launching an attack on protected apps, the adversary often uses static analysis tools to collect some valuable information which contributes to repack apps. These tools typically are disassemblers such as JEB [23] and Apktool [24] that can parse `DEX` file to Java source code, or those such as IDA Pro [16] that can parse `SO` files to ARM instructions.

`JNI_OnLoad`, often regarded as a potential vulnerability by the cracker, primarily exists in all apps as an entry of native layer functions. As shown in Figure 7, when the `JNI_OnLoad` function is debugged in IDA Pro, we can see that the new `.nisl` section abounds with lost of atomic operations for the handler. Therefore it is useless for the cracker to establish a complete logical relationship with static analysis.

C. DYNAMIC ANALYSIS BY MEANS OF UNPACKING TOOLS

As shown in Table 3, up to the present, we have collected six representative shelling tools. Next, the most recent three of them will be chosen as test cases for system resistance.

`DexHunter` [8]

DexHunter is an automatic unpacking tool for Android `DEX` files. The main idea is to fully restore instructions in memory during class initialization. More in details, it directly modifies the Android source codes (Android 4.4.3) and replaces the original content in `art/runtime/class_linker.cc` (ART)

and dalvik/vm/native/dalvik_system_DexFile.cpp (DVM) to own customized codes, to actively load and initialize the classes in all DEX files before system invoking dvmDefineClass.

As above mentioned, without loading and initializing the upper DEX Java layer, the methods virtualized by the CodeCloak system are processed directly at the native layer. It is lower than the DexHunter's unpacking point, so we draw a clear conclusion that DexHunter can not break through the barrier of CodeCloak.

PackerGrind [10]

PackerGrind, a novel adaptive unpacking system, monitors the protected app from the runtime, system, instruction layer then recovers the DEX files according to the collected data. In runtime, PackerGrind tracks the process of parsing DEX files, loading classes, resolving methods, and executing methods. This runtime tracking does not work in the app protected by CodeCloak because our protected app does not have the operation of shelling and DEX restoration. At the same time, some system functions like memcpy(),strcpy() are tracked in PackerGrind to monitor the memory operation, even at the instruction level. Apps protected by CodeCloak, however, do not involve the memory operation of the data in DEX file, so PackerGrind can not monitor the useful data when app running.

DroidUnpack [25]

We further use the advanced unpacking system DroidUnpack for the security evaluation of CodeCloak. As far as we know, DroidUnpack is a powerful reverse analysis tool, it can set up multiple unpacking detection points, including the hidden code extraction, self-modifying code detection, and multi-layer unpacking detection. However, the protection methods involved in CodeCloak do not use the trick of shells, and the scheme does not comprise a set of packing and unloading, so the DroidUnpack's shelling monitoring point is invalid. We should point out that another important function in DroidUnpack is the detection of the JNI reflection interface. Although JNI related API calls can be detected in the protected APK file, it is still problematical to restore and modify the instructions due to diversity virtualization, not to mention repackaging.

D. MANUAL DYNAMIC ANALYSIS

We have performed static and dynamic analysis tools to evaluate CodeCloak, and this part will further describe the manual attack process [19] in details.

To intuitively analyze the reverse time cost of protected applications, we first visualize the whole attacking processes as the model shown in Figure 8, then we will apply this model to specify our test case.

At the beginning of an attack process, it is possible for an adversary to encounter some anti-debugging obstacles in $P1$, $P2$. Suppose that the probability of encountering anti-debugging obstacles in $P1$, $P2$ is $p1$, $p2$, respectively; the number of anti-debugging mechanisms is $N1, N2$, respectively. Assume that the time required for an attacker to pass

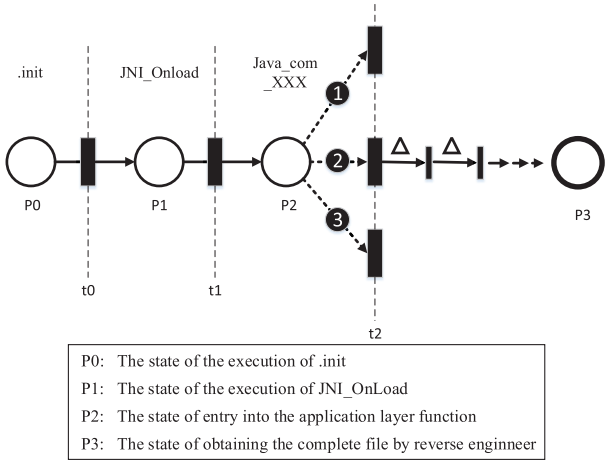


FIGURE 8. A formal model for manually reverse attack was proposed. In this model, from *P0* to *P3*, it indicates several states that occur during manual dynamic analysis of protected Android applications.

an anti-debug is Tx, and the state after $P2$ is to enter a state of the virtual machine, in other words, the method we protect is the Java_com_XXX method. Since diversity protection is introduced here, an attacker may randomly select one of the three paths during the attack. We assume that the attacker chooses path 2 here, and assume that the attacker needs Tt time to find the mapping between a virtual bytecode and the original instruction. So the time required to get to the state $P3$ is as follows:

$$t1 = t0 + p1 * (Tx * N1), \quad (0 < p1 \leq 1), \tag{1}$$

$$t2 = t1 + p2 * (Tx * N2) + (Tt^n)^m, \quad (0 < p2 \leq 1), \tag{2}$$

where n represents the number of virtual instructions; m represents the number of mapping tables between virtual instructions and original instructions.

Therefore, the total time cost will be as follows:

$$T_{all} = t0 + (p1 * (Tx * N1)) + (p2 * (Tx * N2)) + (Tt^n)^m, \tag{3}$$

It can be seen from equation 3 that the total time cost required by the attacker is equal to the time cost of bypassing the normal anti-debugging plus the time cost of breaking the virtual machine. Due to the introduction of the diversified virtual machine principle, the attacker's attack cost increases exponentially. If we also virtualize the JNI_OnLoad and Java_com_XXX methods at the same time, $t1$ and $t2$ will increase more significantly.

After a short discussion of model formation, specifically, we will evaluate the security of our protection scheme through a certain scale of attack experiments. Participants in the program are 22 students from the host institution who are pursuing a degree in computer network security. Among the 22 students, 20 are masters and 2 are doctors. 10 of them are women and 12 are men. The 22 attackers need to complete the following three tasks within the specified 48 hours:

TABLE 4. Number of completions and average time cost per task.

	Task 1	Task 2	Task 3
Completion number[1]	18	9	3
Time cost[2]	7.5	9.9	29.3

[1] In this table, the completion number indicates the number of people who successfully completed each task.

[2] In this table, the time cost indicates the average time cost of successfully completing each task. The unit of time cost is an hour.

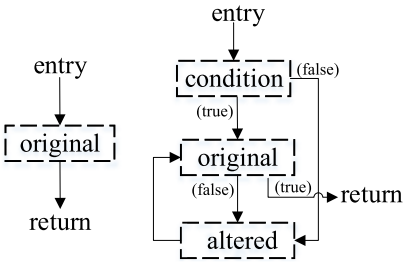


FIGURE 9. Data flow diagram before and after confusion by OLLVM. The entry is the native swipeRight method, and the altered refers to bogus logic.

`Task 1`: Given conventional anti-debugging and obfuscation mechanisms, find the entry point address of the real VM interpreter.

`Task 2`: Given the entry address of the VM interpreter, find the entry address of the scheduler.

`Task 3`: Given the scheduler's entry address, find and record the order, address, and content of handlers.

The three tasks describe the process that an adversary will face when attempting to complete attack steps. The fewer the number of task people completes, the more reliable the protection of the solution performs. In addition, in order to specify the level of attack difficulty, we count the average time that attackers ended each task. In a word, the longer each task takes, the more difficult CodeCloak makes it for an attacker. Table 4 shows the completion of each task.

The table shows that eighteen volunteers have completed `task 1`. Since all participants have some reverse experience, it is easy to find the entry address of the real VM interpreter. However, there are only nine volunteers to complete `task 2` because the virtual machine's dynamic scheduling mechanism makes it difficult for them to track. We apply the existing detection algorithms [26], [27] to measure the ARM instructions similarity between the handlers extracted by each volunteer and the pre-protection code segments. We record the total number of handlers collected in `task 2` and `task 3`.

As shown in Table 5, most volunteers cannot extract and restore the complete instruction set. Only three volunteers extract all the handlers with over 95% instruction similarity. We must point out that many volunteers mistake that they obtain all the instruction sets. In fact, only part of those

TABLE 5. The number of handlers collected by different attackers and the similarity of instructions.

Attacker	P1	P2	P3	P4	P5
(Hn,Sn)[1]	(7829,0.31)	(0,0)	(23496,0.96)	(5829,0.24)	(7834,0.32)
Attacker	P6	P7	P8	P9	
(Hn,Sn)[1]	(15634,0.60)	(23487,0.95)	(15675,0.66)	(23510,0.98)	

[1] This table shows the experimental data (Hn, Sn) of nine attackers (P1-P9) who successfully found the scheduler entry address in Task 2. where Hn is the number of all handlers collected by the attacker, and Sn is the similarity between the ARM instructions of the extracted handlers and the ARM instructions of the code segment to be protected. Zero data indicates invalid data.

instructions could be got because of multiple sets of virtualization mechanisms(our configuration here is `3`). At last, we consider that three volunteers could complete all the attack tasks.

These results lead to a further study finding more details about time cost. We delved into the average time cost and the reason why some volunteers can complete each task. Table 4 shows that the average time cost of anti-debugging by 18 attackers is 7.5 hours. The average time cost of breaking the virtual machine mechanism is the sum of 9.9 hours to complete task 2 and 29.3 hours to complete task 3. Considering that most people do not complete `task 2` and `task 3`, the real time cost will only be greater.

E. VIRTUAL SPECIFIC ANALYSIS

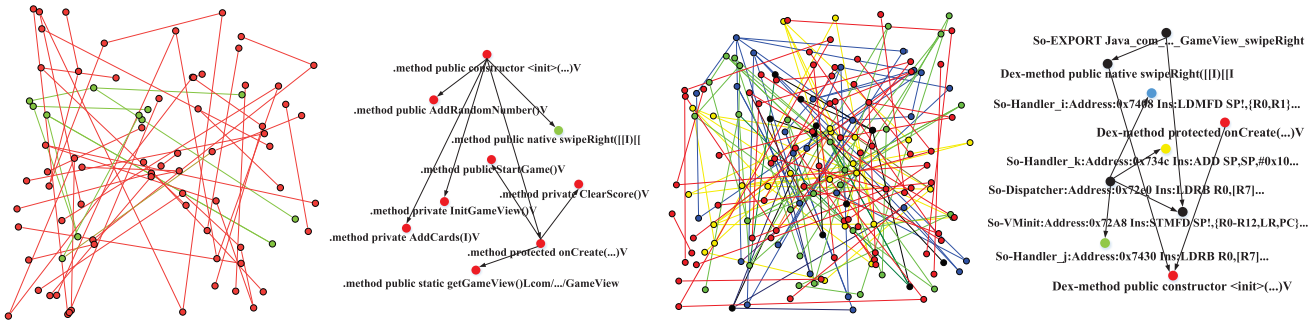
In the past few years, code obfuscation based on virtualization has exhibited a general trend in software protection. However, at present, some well-designed attacks also involve breaking these protections. It can be viewed at least from the following two aspects.

1) REVERSE ENGINEER VM-BASED PROTECTION

In the case of CodeCloak protection, we take Rolles's scheme as a typical example to evaluate security against reverse attack. As an adversary, he/she will try his/her best to obtain code execution and data processing of `VMSection`.However, the diversity of virtualization makes it very difficult to crack and restore protected APKs. We must point out that this kind of attack method for virtualization protection is based on the process of interpreting the execution of bytecodes. So a conclusion can be easily drawn that attackers must be familiar with the principle and structure of the entire code virtualization protection in advance. This means, in simple words, after parsing and semantic analysis of `VMSection`, an adversary still needs to generate the platform-dependent machine code that is similar to the pre-virtualized code. And the diversity of the CodeCloak protection process will defend virtual machine attacks.

2) VM REPLACEMENT ATTACK

The key of the scheme Sudeep Ghosh is to replace the `VMSection` module of the protection program with



(a) DFD of original 2048.apk (b) Partial magnification of (a) (c) DFD of protected 2048.apk (d) Partial magnification of (c)

FIGURE 10. Using multi-virtualization protection or not: (a) The red nodes and the green nodes respectively refer to the APIs of the DEX layer and the native layer's ARM instruction blocks containing jumps and call relationships. (c) The green, blue, and yellow nodes refer to the native layer's ARM instruction blocks generated by CodeCloak's first, second, and third protections, respectively. The red nodes and the black nodes refer to the APIs of the DEX layer and the same instruction blocks generated after three protections.

the attack VM component. Once accomplishing this step, an adversary could analyze the dynamic running state of the application in memory. In fact, the basic premise behind this attack strategy is that the VMSection is not anchored sufficiently to the execution environment, that is to say, the VM needs to exist in the running environment as a single module. However, as mentioned above, VMSection in CodeCloak-protected application is embedded in SO files rather than independent. Therefore, the high coupling relationship between VMSection and SO files effectively prevents this attack method above.

F. COMPLEXITY ANALYSIS

1) DATA FLOW DIAGRAM ANALYSIS

Complexity in the data flow diagram (DFD) is critical to evaluate the security. In this section, we utilize the application function call graph before and after protection. Applying Androguard [28] to get the function call graph gexf of the DEX layer for 2048.apk [22], we run the python script to analyze the APIs of a protected code segment. Use IDA Pro [16] to record the native layer call relationship manually and then build the block call graph gexf, finally use the Gephi tool [29] to parse the gexf files.

Fig 10 (a) shows the result of parsing gexf for the native layer and the DEX layer before protection. DEX APIs are displayed as red, and the native layer's ARM instruction blocks containing jumps and call relationships are marked in green. There are a total of 93 DEX layer APIs and 12 native layer blocks. Figure 10 (b) give a partial detail view of Figure 10 (a) with some tags. Figure 10 (c) is a schematic diagram of the analysis results after protection by CodeCloak. The yellow, green, and blue nodes describe several ARM instruction blocks that contain jumps in the native layer after three executions, 40 blocks of native level are generated in each protection round. Compared with the simple bogus control flow and control flow flattening of the OLLVM [13] shown in Figure 9, there are almost completely different blocks calling graph in every protection round, for an adversary, must plan well, get information, finish one attack at a time; or she/he should know every possible running blocks sequence. Without doing this a successful attack cannot be guaranteed.

2) DIVERSITY EVALUATION

One of CodeCloak's design goals is to increase the diversity of program execution. To evaluate this goal, we record the offset addresses and orders of the first hundred handlers called at each run. The intuition behind this planning is that if the program has different execution address offsets per round, an adversary would hardly complete tracking and debugging. Applying 2048.apk [22] as a test program, we run the protection system ten times. In order to collect dynamic location information, we use IDA Pro [16] to debug the protected application and manually collect the addresses of the handlers.

As shown in Figure 11. For one protection, the 100 handler offset addresses collected have changed a lot. However, for ten protections, only 83 of the 10,00 nodes are entirely overlapped. We can quickly draw reasonable conclusions: CodeCloak-protected codes exhibit strong non-deterministic behavior in one run; The offset addresses of the handlers at each runtime are basically different. In other words, it is difficult for an attacker to use the previously collected runtime information to perform reverse engineering.

G. PERFORMANCE EVALUATION

In this section, we evaluate the overhead of CodeCloak in terms of time and space complexity. The time complexity includes the startup time of the apps, and the space complexity consists of the size of the apps and the overall memory consumption at runtime. In order to make the experiments more convincing, we obeyed the following principles in the selection process of the test apps: Firstly, we selected the apps containing the binary SO file as much as possible, which provide protected objects for the CodeCloak system. Secondly, the selected apps must be popular enough, and each app has more than 1,000,000 downloads in Google Play. Finally, the selected apps must cover most of the application

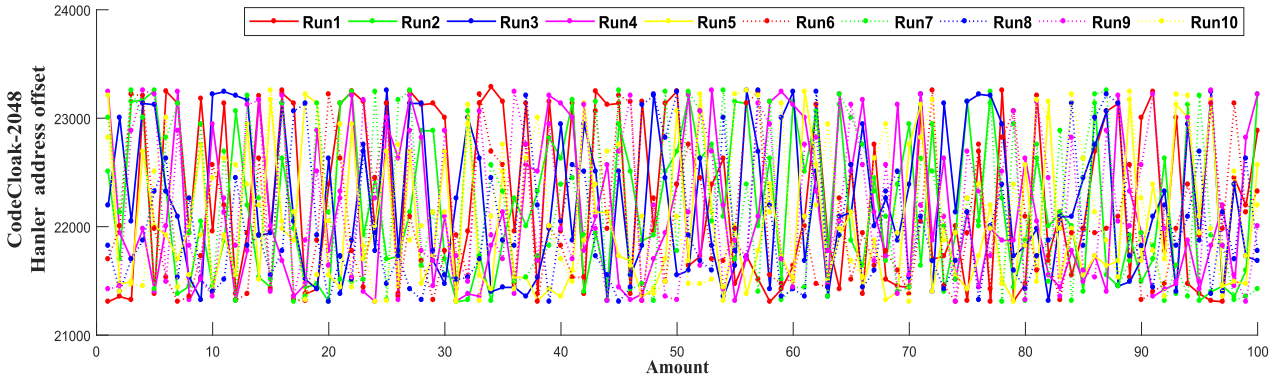


FIGURE 11. Time diversity after protection by CodeCloak. The horizontal axis represents the scheduling order of the partial handlers, and the vertical axis represents the position offset of the handler.

TABLE 6. Information about App collection set.

App Name	Category	App Name	Category
Dual Space	Tools	American Airlines	Travel
2048	Games	Tumblr	Social
Lifelog	Health&Fitness	Yahoo Sports	Sports
Oxford Dictionary	Education	Paper Camera	Photography
Voice Changer	Entertainment	Fox News	News&Magizines
Yandex Weather	Weather	Launcher IOS 12	Personliazation
Photo Video	Music&Video	Calculator Pro	Business
HDplayer	Media&Video		

categories defined by Google Play, such as tools, games, etc. Details about them are shown in Table 6.

Our experimental platform is a Google Nexus 5 smartphone with Android version 4.4.2. We use Tencent's GT tool [30] for testing. The work of DSVMP [19] suggests that the performance overhead of 5 VM configurations is moderate in a virtual machine protection system, so we use CodeCloak with 5 VM configurations in our experiments.

The size of the protected APK is an important indicator. With that general outlook, it follows that if a protected program increases the size heavily, the system CodeCloak will be limited in many applications. As can be seen from Figure 12, the protected APK size has increased by an average of 23.79%. That is because `VMSection` is embedded in the `ELF` file. At the same time, we found that the growth rates for different APK sizes vary widely. For example, `2048.apk` [22] increased by 42.80% but `Lifelog.apk` [31] by 11.49%. The reason is that the growth of each APK is only related to `VMSection`. So we think that if the original program is smaller or approximately the same size as `VMSection`, it could be the high growth rate. In another word, if the size of the APK itself is small, the volume of the APK caused by `VMSection` will increase significantly, and vice versa. The volume of `2048.apk` and `Lifelog.apk` before protection are 2.9MB and 17.0MB respectively. The overall volume increase is acceptable relative to the size of the application itself, and the size of the APK can be reduced by a series of slimming methods.

Next, we evaluate how CodeCloak affects memory consumption and perform 100 protections in each sample. The result shows that the memory consumption after virtualization increases by an average of 17.61%. It is acceptable relative to the importance of the algorithm in the program.

Figure 12 shows overhead of application start time ranges from 1.28% to 7.33%, with an average of 4.08%. All protected apps start in less than 1 second on our platform, which is modest.

Due to the different protection objects, we compare performance evaluations with two commercial systems involving `SO` protection: UPX shell [11] and OLLVM-based Hikari [32] rather than DIVILAR [9]. Table 7 shows that in terms of volume growth, the Hikari-confused apps have a significant code expansion rate up to 50%, but CodeCloak only has slightly increased the size of the protected code segment compared to its peers. As for memory consumption, the average cost of Hikari-confused apps is 1.52 times that of the original programs, while the UPX shell is similar to CodeCloak. In the startup time, the CodeCloak with 5 VMs performs better than the other two protection systems. Different from common packer protection, CodeCloak has no process of decryption and unpacking at startup. Therefore, less time is spent processing load. As a result, the time complexity and space complexity perform better to some extent.

V. RELATED WORK

Research in this field mainly focuses on the detection and preventing of repackaged apps. AdRob [33] is the first large scale study on the characteristics of cloned mobile applications and their impact on the original developers. Repackaging detection based on code similarity includes [34]–[37]. Other malicious behavior detection methods include Apposcopy [38], NDroid [39] and [40].

In repacking prevention, obfuscation, packaging, and encryption are often used, but they can just defend against static analysis instead of dynamic reverse engineering [41]. In addition, AppInk [42] uses software watermark technology

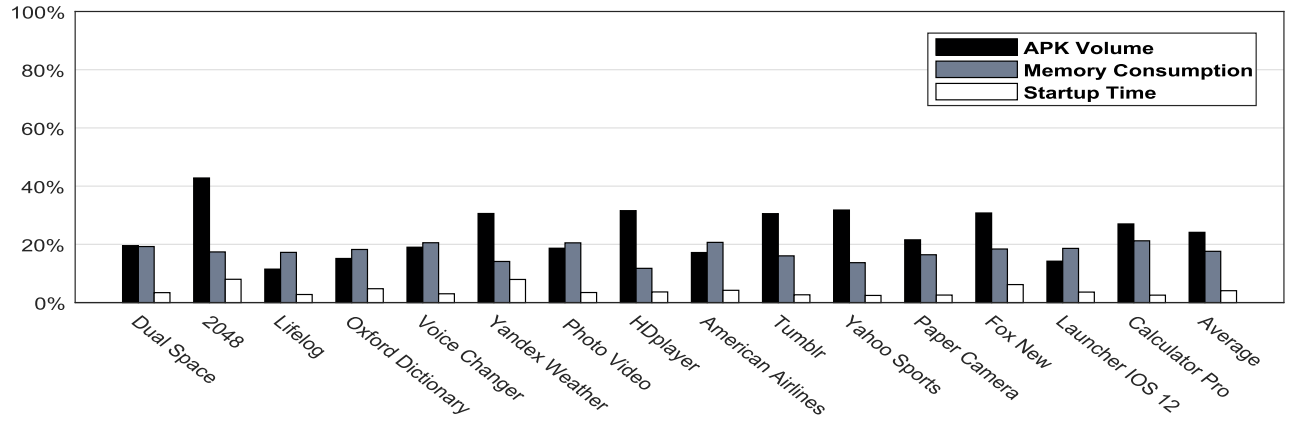


FIGURE 12. This figure shows the growth rate of file size, memory, and startup time of several apps protected by CodeCloak.

TABLE 7. Comparing with the average performance of apps protected by the prior work.

Performance[1]	Original	CodeCloak	UPX shell	Hikari
APK Volume (MB)	12.05	14.91	11.39	18.44
Memory Consumption (MB)	47.60	55.97	52.98	72.35
Startup Time (ms)	624.50	649.98	874.3	886.79

[1] Performance includes average APK volume, average memory consumption, and average startup time for 15 apps before and after protection.

to tame app repackaging. Converting Java layer code to more complex native C/C++ implementations is more advanced [43]. Reference [44] analyzes how different methods of protection, namely class encryption and usage of native code, affect decompilation of Android apps. Wu Zhou converted Davilk instructions into virtual instructions, and explained the instructions by Hook mechanism [9].

An increasing number of developers usually put the critical logic in the native shared library in C/C++ implementation, the security of SO file needs to be solved urgently. Native code obfuscation based on LLVM compiler [13] makes it difficult for reverse engineering, but source-level processing is often difficult to operate, and other OLLVM-based approaches [45], [46] have the same problem.

VI. CONCLUSION

In this paper, we introduce CodeCloak, a new method of native ARM instruction virtualization protection based on time diversity. It can effectively resist the threat of deep repackaging attacks. As far as we know, CodeCloak is the first system to take advantage of virtualization technology to protect native SO files.

Our evaluations show that CodeCloak can effectively resist static analysis, dynamic analysis, and even specific attack methods. Manual dynamic analysis experiments show that the attack costs of malicious attackers increase exponentially due to the introduction of diversified virtual machine principles. Performance experiments show that CodeCloak brings a small increase in performance overhead, which is acceptable relative to the importance of the algorithm in the program.

ACKNOWLDEGMNENT

(Zhongkai He and Guixin Ye are co-first authors.)

REFERENCES

[1] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2017.
[2] X. Jiang and Y. Zhou, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 95–109.
[3] P. Yong, L. Jie, and L. Qi, "A control flow obfuscation method for Android applications," in *Proc. Int. Conf. Cloud Comput. Intell. Syst.*, Aug. 2016, pp. 94–98.
[4] (2012). *Proguard*. [Online]. Available: https://www.guardsquare.com/en/products/proguard
[5] (2017). *Dexguard*. [Online]. Available: http://www.saikoa.com/dexguard
[6] (2015). *Dexextractor*. [Online]. Available: https://github.com/lambdalang/DexExtractor
[7] Jack Jia. (2014). *Android App Dynamic Reverse Tool Based on Xposed Framework*. [Online]. Available: https://github.com/halfkiss/ZjDroid
[8] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed Android applications," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2015, pp. 293–311.
[9] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform," in *Proc. ACM 4th Conf. Data Appl. Secur. Privacy*, 2014, pp. 199–210.
[10] X. Lei, X. Luo, Y. Le, W. Shuai, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, May 2017, pp. 358–369.
[11] Fish_Ou. (2015). *Android SO UPX*. [Online]. Available: https://www.cnblogs.com/fishou/p/4202061.html
[12] GitHub. (2014). *UPX Shell Tools*. [Online]. Available: https://upx-shell.en.softonic.com/#
[13] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, May 2015, pp. 3–9.
[14] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. ACM 22nd SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 757–768.
[15] Currwin. (2015). *DecLLVM*. [Online]. Available: https://github.com/F8LEFT/DecLLVM
[16] Hex-Rays. (2015). *IDA Protect*. [Online]. Available: https://www.hex-rays.com/index.shtml
[17] iJiami. (2014). *iJiami SO Protect*. [Online]. Available: http://www.ijiami.cn/soProtect
[18] T. Legu. (2014). *Tencent Legu*. [Online]. Available: http://legu.qcloud.com/

[19] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and W. Zheng, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling," *Comput. Secur.*, vol. 74, pp. 202–220, May 2018.
[20] J. Seward. (2013). *Valgrind*. [Online]. Available: http://valgrind.org/
[21] Oreans. (2015). *Oreans-Technology*. [Online]. Available: https://www.oreans.com/codevirtualizer.php
[22] Google Play. (2018). *2048.apk*. [Online]. Available: https://play.google.com/store/apps/details?id=com.androbaby.game2048
[23] PNF Software. (2015). *JEB*. [Online]. Available: https://www.pnfsoftware.com/
[24] R. Wisniewski. (2010). *Apktool*. [Online]. Available: https://ibotpeaches.github.io/Apktool/
[25] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, and X. Wang, "Things you may not know about Android (un)packers: A systematic study based on whole-system emulation," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2018, pp. 18–21.
[26] L. Luo, M. Jiang, D. Wu, L. Peng, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. ACM 22nd SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 389–400.
[27] Z. Đurić and D. Gašević, "A source code similarity system for plagiarism detection," *Comput. J.*, vol. 56, no. 1, pp. 70–86, 2013.
[28] Jbremer. (2015). *Androguard*. [Online]. Available: https://pypi.org/project/androguard/3.0/
[29] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," in *Proc. AAAI 3rd Int. Conf. Weblogs Social Media*, 2009.
[30] Tencent. (2016). *Gttool*. [Online]. Available: https://github.com/TencentOpen/GT
[31] Google Play. (2018). *Lifelog.apk*. [Online]. Available: https://play.google.com/store/apps/details?id=com.sonymobile.lifelog
[32] Tencent. (2017). *Hikari*. [Online]. Available: https://github.com/HikariObfuscator/Hikari
[33] C. Gibler, R. Stevens, J. Crussell, C. Hao, Z. Hui, and H. Choi, "AdRob: Examining the landscape and impact of Android application plagiarism," in *Proc. Int. Conf. Mobile Syst.*, 2013, pp. 431–444.
[34] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 56–65.
[35] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2012, pp. 317–326.
[36] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable detection of semantically similar Android applications," in *Computer Security—ESORICS*. 2013.
[37] W. Zhou, "Repackaged smartphone applications: Threats and defenses," Ph.D. dissertation, GradWorks, 2013.
[38] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. ACM 22nd SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587.
[39] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, "On tracking information flows through JNI in Android applications," in *Proc. IEEE/IFIP 44th Annu. Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 180–191.
[40] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.
[41] K. Lim, Y. Jeong, S.-J. Cho, M. Park, and S. Han, "An Android application protection scheme against dynamic reverse engineering attacks," *JoWUA*, vol. 7, no. 3, pp. 40–52, 2016.
[42] Z. Wu, X. Zhang, and X. Jiang, "AppInk: Watermarking Android apps for repackaging deterrence," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Secur.*, 2013, pp. 1–12.
[43] M. Protsenko, S. Kreuter, and T. Müller, "Dynamic self-protection and tamperproofing for Android apps using native code," in *Proc. 10th Int. Conf. Availability, Rel. Secur.*, Aug. 2015, pp. 129–138.
[44] S. Ilić and S. Đukić, "Protection of Android applications from decompilation using class encryption and native code," in *Proc. Zooming Innov. Consum. Electron. Int. Conf.*, Jun. 2016, pp. 10–11.
[45] K. Lim, J. Jeong, S.-J. Cho, J. Cho, M. Park, S. Han, and S. Jhang, "An anti-reverse engineering technique using native code and obfuscator-LLVM for Android applications," in *Proc. ACM Int. Conf. Res. Adapt. Convergent Syst.*, 2017, pp. 217–221.
[46] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2014, pp. 25–36.

ZHONGKAI HE received the B.E. degree in computer science and technology from Northwest University, China, in 2017. He is currently pursuing the M.S. degree in computer application technology with the School of Information Science and Technology, Northwest University, Xi'an, China. His research interests include android security and deep learning.

GUIXIN YE was born in Taian, Shandong, China, in 1990. He is currently pursuing the Ph.D. degree in software engineering with Northwest University. His research interests include privacy and software security. He has extensive experience in authentication and software bug detection.

LU YUAN was born in 1995. She received the B.S. degree in computer science from Northwest University, China, in 2017, where she is currently pursuing the master's degree. Her research interests include mobile computing and mobile energy efficiency.

ZHANYONG TANG received the Ph.D. degree in computer software and theory from Northwest University, where he is currently an Associate Professor with the School of Information Science and Technology. His research interests include network and information security, software security and protection, localization, and wireless sensor networks.

XIAOFENG WANG received the M.S. and Ph.D. degrees from the School of Information Science and Technology, Northwest University, in 2005 and 2008, respectively. From 2002 to 2005, she was a Tutor with the School of Information Science and Technology. Since 2005, she has been an Assistant Professor with the College of Information Science and Technology, Northwest University. Her research interests include pattern recognition, image processing, multimedia processing, and data mining.

JIE REN was born in 1988. He received the Ph.D. degree in computer architecture from Northwest University, China, in 2017. He is currently an Assistant Professor with the Computer Science Department, Shaanxi Normal University. His current research interests include mobile computing and performance optimization.

WEI WANG received the B.S. and M.S. degrees in communication engineering from Xidian University, Xi'an, China, in 2000 and 2004, respectively, and the Ph.D. degree in information and communication engineering from Northwestern Polytechnical University, Xi'an, in 2017. Her research interests include localization algorithm, tour planning of mobile node, and cross technology communication.

JIANFENG YANG received the B.S. degree in electronic information science and technology and the M.S. degree in computer science from Northwestern University, Xi'an, Shaanxi, China, in 2003 and 2009, respectively. His research interests include the monitoring of the development level of educational informatization, and the construction of university informatization.

DINGYI FANG received the Ph.D. degree in computer application technology from Northwestern Polytechnical University, Xi'an, China, in 1983. His current research interests include mobile computing and distributed computing systems, networks and information security, and wireless sensor networks.

ZHENG WANG received the Ph.D. degree in computer science from the University of Edinburgh, in 2011. He is currently an Academic Advisor with C43, Infolab21, Lancaster University, Lancaster, U.K., under the supervision of Prof. M. O'Boyle. His current research interests include parallel compilers, runtime systems, and the application of machine learning to tackle the challenging optimization problems within these areas.