Protection Android App with MultiDEX and SO Files from Reverse Engineering

MA Rahim Khan[1†] and Manoj Kumar Jain[2††],
khan_rahim@rediffmail.com *manojjain@lingayasuniversity.edu.in*
Department of Computer Science and Engineering, Lingayas Vidyapeeth, Faridabad, Haryana India

Abstract
Reverse engineering technique gives the attacker the gamble to embed the new code with apps; there is a chance of losing confidential information or adding some adware. Some most popular tool protects from reverse Engineering like LIAPP, IJIAMI, and ALIBABA, etc. Those tools can encrypt the DEX file in APK; before loading the files dynamically, it decrypts the files; the MultiDEX files do not defend these tools. We propose an approach to protect from reverse engineering methods. Our aim to protects against static repackaging attacks and provides better efficiency in decrypts and loading apps. We introduce the Virtual Machine (VM)-based MultiDEX and share object (SO) protection approach; we used the newly stack-based native code system in this approach. It provides intense level protection under the virtual machine. It leverages multiple dynamic virtual machines protection of bytecode to minimize the overhead. The significant advantage of our approach to protecting the MutiDex files and SO files is that very few systems are available, which take care of both types of files.

Keywords:
MultiDEX files, Reverse Engineering, multiple virtual machines, AES encryption, SO files.

1. Introduction

The repacked app, the attacker, takes a benign and genuine app and injects malicious code in the source. Again, repacked the app for distribution is the most of the previous approach used for hijacking benign profits and committing other malicious acts to achieve financial or non-financial objectives. Additionally, third-party apps are weaker than Google play stores; as per one statics survey of 2020, [1] 1.7 million users are infected from malicious apps. Some Android devices do not use the support of Google Play, using the third-party apps store. Many third-party apps are not trustworthy, found that most apps are repackaged apps embedded with malicious code and published on third-party apps stores.

A recent survey shows that 95% app of games is repackaged [2]. App repackaging is especially notorious due to financial loss and honest developer, but its terrible impact on the total app ecosystem and users. Most attackers download the app from the various site, then repackaged the app with malicious code with their name to earn the purchase profit or embed ad library, causing the ad's benefit to attackers [3]. Therefore, some techniques can protect reverse engineering and protect the source code from embedding malicious code. Code Obfuscation is an essential method to protect the android application against reverse engineering and repackaging.

Table 1. Code protection scheme from different attacks

Type of File	Method	Counte-ractant	Set Threshold	Core dump	Other operation
DEX Files	DEX Encryption	DexExtractor[4]	memcpy	Get the Complete source code from Dex File	-
DEX Files	DEX Extractor	ZjDroid[5], DexHunter[6]	DvmDefine class	Get the Complete source code from Dex File	-
DEX Files	DIVILAR[7]	PackGrind[8]	memcpy	Get the Complete source code from Dex File	Get the key, repackage the according to the mapping table
SO Files	UPX Shell[9]	UPX Shell Tool[10]	Init/.init_array	Get the dump file	Fix the fields, open with IDA
SO Files	OLLVM [11]	DEC LLVM[12]	Init/.init_array	Get the dump file	Decrypt JNI, editing into source code
Multi DEX and SO Files	MultiDex and SO protection	DexHunter[6], Packer Grind [8], DroidUnpack [26]	memcpy Init/.init_array	Unable to get source code from MulDex Files	-

In Tab 1., previous works are summarised, multiple approaches goal at the DEX bytecode level. Two methods, such as DexGuard[13] and ProGuard[14], are used to protect the DEX files in Android App, but those methods cannot protect the obfuscated code if the tool like DexExtractor[15] finds the entry point. The attacker monitors the standard library call using tools like ZjDroid

Manuscript received November 5, 2024
Manuscript revised November 20, 2024
https://doi.org/10.22937/IJCSNS.2024.24.11.17

and DexHunter, debugging the native code. A DELILAR[7] is a vital technique use to protect against observing the function calls, but one tool, PackGrind[8], can restore the source code by mapping between a function call and native code.

In the current scenario, many android apps are developed on share libraries, written in high-level languages interpret into native instruction. These share libraries are helping to build the apps frequently and quickly used the core algorithm. Therefore, very important to protect the share object (SO) files from reverse engineering. As a present state, very few techniques are available to protect the SO file against reverse engineering, and those techniques are not providing sufficient protection from reverse engineering attacks.

UPX shelling [9] is a technique to protect against reverse engineering, but the UPX tool [10] can launch the attack to observe the function calls. OLLVM [11] is another method for code obfuscation at compile-time, but against this method, we have ani tool such as DecLLVM [12] for reverse engineering.

In this paper, we proposed an approach that will protect the SO files and MultiDEX files. This approach uses the stack base virtualization to secure the algorithm's functionality and logic used in SO files—stack-based virtualization giving the strength to protect the native instruction. We are using the multiple protection schemes that dynamically choose at runtime. In the next phase, we are implementing the MultiDEX protection scheme using the code obfuscation method. There are very few techniques used for protecting both types of protection in a single approach—our result protecting from reverse engineering and repackaging.

To avoid reverse engineering, some developers using the obfuscation technique to prevent the application code, but the obfuscation technique gives high securities to protect the source code. Source code transforms into a different form [15].

Dynamic scheduling scheme for Virtual Machine-based code obfuscation to protect the MultiDex files of the android app. To increased diversity of code obfuscation through applying the multiple VMs. Stack-based virtualization to protect the SO file.

Section 2: introduce the MultiDex file, VM base code obfuscation, VM components, SO protection, diversifying. Section 3: Describe the Proposed approach. Section 4: Multiple VM techniques. Section 5: Case Study. Section 6: Evolution. Section7: Result Analysis. Section 8: Conclusion and Future works.

2. Background

2.1 MultiDex Files

Meanwhile, some tools (Ijiami and Alibaba) are useful for protecting the source code by using encrypt DEX (Dalvik Executable) [16], stub DEX File moves to under root directory APK, whenever the app is executed, DEX stub decrypt, dynamically loading. However, we found some weaknesses in the tools [17]. It never encrypts all the class files of DEX, shown in Fig.1.

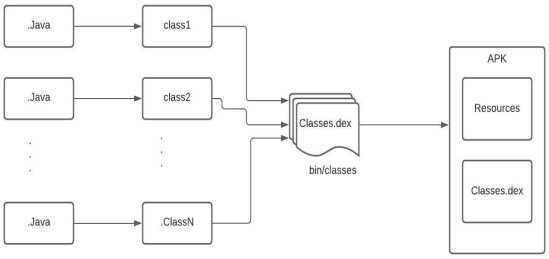

Figure 1: File structure of APK

If any app has more than 65536 methods, apps need to create the multi DEX file; those tools never encrypted entire classes.dex files, tools have the facility to encrypt only one class file. However, apps have the multiDEX file, shown in Fig. 2, except one, all other classes.dex file will be unencrypted. Fig. 2. showing the app having multiple *.dex files.

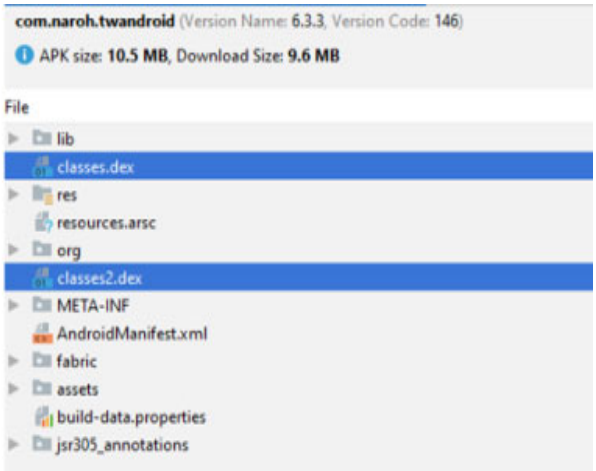

Figure 2: Directory structure of APK with MultiDEX files

A proposed dynamic methodology is used to encrypt all *.dex files [18]. The proposed method supports the Multidex library (Developed by Google). This approach never encrypts/decrypt all *.dex files and resource files into the Dalvik Virtual Machine (DVM). Since the app's execution decrypted all *.dex files and resource files and

dynamically loaded it, attackers can do decryption and do reverse engineering on all DEX files to find the apps source code; the decryption technique creates too much burden during the translation time.

Table 2: Different type of APK have the MultiDEX files

Name of APK
2048_v4.6.2.com.apk
Tiktok_v17.8.4.com.apk
Snapchat_v11.4.5.73.com.apk
Netflix_v7.80.0 build 10 35175.com.apk
Viber messenger calls group chats_v14.1.0.16.com.apk
Shan koe mee skm777_v1.1.com.apk
Golden card games tarneeb trix solitaire_v20.1.0.27.com.apk
Videobuddy youtube downloader_v1.36.136000.com.apk
Waze gps maps traffic alerts live navigation_v4.68.1.0.com.apk
Watched multimedia browser_v1.0.2.com.apk
Nvidia geforce now_v5.33.29272392.com.apk
Yahoo mail – organized email_v6.14.1.com.apk
Kinemaster video editor video maker_v4.15.9.17782.gp.com.apk
Mx player_v1.24.6.com.apk
Textnow free texting calling app_v20.42.0.2.com.apk
Discord talk video chat hang out with friends_v46.3.com.apk

Our proposed methodology decrease difficulties in terms of efforts and time for the attacker to perform reverse engineering—our aim to present a code obfuscation mechanism for share object(SO) Files at the binary level. Stack base virtualization mechanism protect algorithms logics and protocol implemented into SO files. Stack-based virtualization to save native ARM instructions to increased security strength at the machine code level uses multiple virtual protection schemes [19]; at run time, dynamically method is chosen. Our approach protects the MultiDEX files and SO files. We evaluate our method under various repackaging tools; the result showed that our process protects the android app from various repackaging attacks, and it reaches this in minimum overhead. Our first achievement is to protect SO files, and second achievement binary code virtualization for ARM instruction.

2.2 VM based Multidex Android app Protection

The VM protection scheme has some steps to protect the MultiDEX android app. The first target to decompile binary SO file and gather all ARM instruction giving to the pre-set tag. All ARM instruction to mapped to virtual instruction(VI), which similar to ARM instruction. Second target to convert the Virtual instruction into a binary SO file to follow the norm of encoding. Lastly, united the custom interpreted is enclosed in the SO binary file. The VM-based process creates a high cost to the attacker to perform reverse engineering.

To demonstrate the operation, used the DIVILAR [7] as an illustration. DIVILAR provides the VM-based protection technique. It translates the actual instruction set into the virtual instruction set and uses a hook method to restore virtual instruction and interpret it at runtime. It is a beneficial technique to protect the DEX file. VM based protection is useful in static and dynamic analysis.

Our Multidex obfuscation and SO files protection more effective than the DIVILAR, because our approach protects the lower level of SO files and multDEX. DIVILAR object to safeguard the DEX file. The major drawback of DIVILAR design, it used the hook technique to communicate within components. However, an attacker can benefit from a hook mechanism to obtain information whenever the instruction translation happens. Our approach never accepts the restoration process when the program is translated.

Code virtualization is an emerging approach for obfuscation; it builds upon VM (Virtual Machine) and safeguards the program from unauthorized penetration. It is state of the art, and effective VM-based protection methods use a fixed scheduling framework wherever this system always uses just one, deterministic performance course for the same input. Nevertheless, such methods are vulnerable when the enemy may recycle information removed from the formerly observed process to break applications secured with the same obfuscation scheme.

2.3 VM Components

Our method applies the stack-based VM execution, involving various components in the proposed methodology in Fig. 3. The native program background, which has some local variable, a function parameter, return values, address, etc., will be saved into a register-based VM memory location known as VMContex. When VM passes into the system, the native context program is saved and Initialized by the VMInit components. After the execution of protecting the code, VMExit will bring back the native program context. Restore program control return the original program to execute the native machine code.

As per our approach, VM heart consists of Dispatcher and handler; Dispatcher fetch bytecode that is ready for execution interprets the fetched bytecode and gives the handler translation into native machine code the fetched bytecode.

This process will continue till all bytecode translates into specific code segments are executed. The key point is to know the protected code segment logic from the

attacker's perspective and find how the bytecode is mapped into native machine code.

2.4 SO Protection

As mention in Tab. 1, we observed that the SO file also more effective than the MultiDex files. However, we want to know that the existing method could protect authentically or not? We will continue some manual try to attack to understand protection methods in SO files. We are here trying to apply the disassembler IDA Pro [20] to analyze the SO files.

At present, a secure SO file uses encryption; as we know, the UPX shell disassembles the most common form [9]. The complete instructions are encrypted and error full; it is very puzzling for the challenger to understand the semantic code. However, as a top talented cracker to analyze the packing tool, create the dump point into memory.

OLLVM is a method for obfuscation to hide the control flow of the apps. We select the Tencent Legu method to protect the SO file, SO file disordered by JNI_Onload, attacking target where SO file is disorder. Apply the adversary attack on the target, can decrypt the JNI_onload methods.

Hence, in this, we have an obvious conclusion that OLLVM is more problematic than the UPX shell, but the attacker can penetrate both techniques through dynamic debugging.The experiment of paper VM obfuscation can prevent SO file from both static and dynamic debugging attacks, and even avoid any attack on VM.

2.5 Attack Tool

In present research [21] has demonstrated the reverse step of the VM Protection program. Here are the following steps

Step 1: The first main target to find the initial point from VM Interpreter;

Step 2: Find the correlation between the bytecode and corresponding handler function

Step 3: Apply the first both two-step to recover the logic and target code section. These attacking steps are a primary task to perform attacks.

Assume in the system; the attacker has very skillful, talented, hands-on experience in reverse engineer expertise uses the safeguard program multiple times to protect the apps. Also, the attacker can use IDA- Pro, Valgrind[22] tools to modify the SO file in memory, do restore and track.The attacker aims to launch the attack based on the VM working scheme and conversion logic between instructions. Our objective is to secure the VM Working mechanism and mapping scheme between instruction.

3. Methodology

3.1 Design objective

Our proposed approach gives a high level of abstraction and protection from various types of reverse engineering attacks. The software runs on the same execution track in the first setup, it will run several times, and the attacker will learn or obtain appropriate knowledge of the program behaviors. In the second setup, the software runs on different execution tracks. It will run several times; an attacker will not learn or obtain sufficient knowledge of the program behaviors. Variety is the key to protecting the software again the dynamic cumulative attacks—our objective diversity of execution for code obfuscation. Our approach will provide the security of apps at a lower and depth level using binary code level. Fig. 3 depicts the whole system of the proposed methodology.

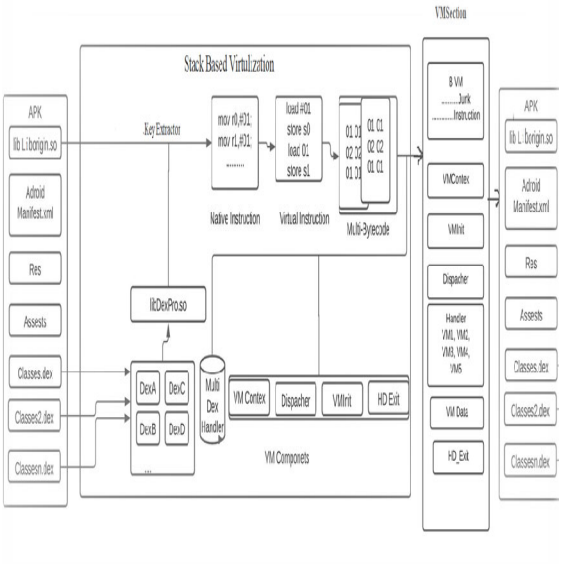

Figure 3: Proposed Methodology

Apk is input into the system; it binds them with the virtualized binary system to interpret the apk as output. Code obfuscation focus on the protection of native SO files and MultiDEX files. We can divide our methodology into different virtualization modules.

3.2 ARM VM Based Model

This protection scheme is based on stack virtualization for ARM instructions. Our prevent approach divided into two segments

I. Multiple virtualizations code transformation
II. Customize interpreter engine

In the first segment, obfuscation rephrases the instruction into a new form, enclosing the mapping table and nuclear handles. First, extract the key and obtain binary instruction from the pre-set code segment from the beginning and end address of apk, then, putting on this mapping scheme, the system could translate the one by one instruction from native to the virtualized format. We commonly employ multiple custom instructions to interpret into a native instruction. We can rapidly conclude that the mapping scheme is essential. To evade being cracked, multiple set schemes and consistent handlers are designed for interpretation instead of the lone map ARM VM. Each time every original instruction can be transformed into virtual instruction with a different code set. In the second segment, obfuscation design is an abstract interpretation engine as VMSecction in a secured SO file. VMSection is known as a code pump; it simulates several schedule operation types on a real CPU. VMSection has various kinds of components.

VMData changed over custom virtual instruction, the Initialization of program VMInit, VMContex designed by register setting, Dispatcher is a virtual machine scheduler. The leave from the system program is HD_Exit and comparing tasks Handlers. VM section interprets into bytecode at run time.

Subsequently, suppose an attacker needs to break this virtualization scheme. In that case, he/she should get a handle on all data to complete translation into virtual instruction. Then, he/she needs to reestablish the original functionality of the running virtual instruction, which is challenging to find the actual instruction from Multiple VM, which is very effortless. There are multiple schedulers to deal with virtual instruction from Bycode instruction.

3.3 Diversifying

Our VM based approach used code obfuscation, which has policies for scheduling multiple instructions, multiple dispatchers handling the single virtual instruction, and bytecode handlers. This approach offers semantically equivalent to a virtual instruction implementing in different ways by various handlers. Thus, the scheduler is dynamically controlled, which of handlers are used to interpret virtual instruction into bytecode. By applying the obfuscation method, multiple sets of handlers are produced. The rule of the handler is obfuscated could differ for a different region of code. Our multiple VM scheme approaches provide multiple VM implementation. Therefore, we obfuscated each handler using the deformation engine. N set of handlers has semantically equivalent with different execution and control flow. Virtual instruction will be translated into a unique bytecode from each VM.

4. Multiple VMs

VM has two types of obfuscation approaches; the first is a single VM (SVM) base; the second is the multiple VM base. Multiple VM (MVM) base is more secure compared to SVM because it has bytecode instruction and set of the handlers. Bytecode instruction manages or schedules by the multiple VM; similarly, virtual instruction is translated by the numerous handlers. Therefore, we can understand various mapping is possible between bytecode instruction and handler. This paper approaches increase the diversity of the program.

4.1 Switching between multiple VMs

This paper can use multiple VM, number VM, and parameters depend on the target program. If we want to increase the program protection, we need to increase the number of VM; multiple VM can create an overhead issue. Therefore, we need to balance the between multiple VM and overhead of the program. This approach for each VM produced a set of handlers. We have N sets of handlers deal with N VMs. Our current scenario interprets the virtual instructions into a handler set; each handler set has two different type bytecode sets. Different bytecode has other VMs so that various handlers translate the different bytecode in different VMs.

Our approach, dynamically Dispatcher decide, which VM in use at run time, Fig. 4, is depicted as Dispatcher. VM switching technique selects one among the multiple VM (Seg. 0X0328 up to 0X0339) update program counter moves to target VM (Seg. 0X033B-0X0353), our approach follows X86 register scheme for the opcode instruction set. The set of bytecode instruction and a set of bytecode handler will be switching aimlessly between different code segments.

```
seg000:00000328
seg000:00000328
seg000:00000328 sub_328         proc near               ; DATA XREF: seg000:00DD8CDC↓o
seg000:00000328                                         ; seg000:00F9E25A↓o
seg000:00000328                 pusha
seg000:00000329                 out     dx, eax
seg000:0000032A
seg000:0000032A loc_32A:                                ; DATA XREF: seg000:00FA35D8↓o
seg000:0000032A                                         ; seg000:00E40B5C↓o
seg000:0000032A                 cmp     bh, [edi+ebx*4+24h]
seg000:0000032E                 pinsrw  mm6, word ptr [ecx+47h], 0DAh
seg000:0000032E                                         ; DATA XREF: seg000:00E88FE8↓o
seg000:00000333                 wait
seg000:00000334                 dec     ecx
seg000:00000335                 push    0FFFFFFE9h
seg000:00000337                 adc     al, 69h ; 'i'
seg000:00000339
seg000:00000339 loc_339:                                ; DATA XREF: seg000:00B6DDA4↓o
seg000:00000339                 cmp     bh, dl
seg000:0000033B                 mov     ebp, 0AF51D757h
seg000:00000340                 cmp     ah, ch
seg000:00000342                 fisubr  word ptr [ebx+68h]
seg000:00000345                 fmul    dword ptr [edi-60E28ECFh]
seg000:0000034B                 outsb
seg000:0000034C                 push    esp             ; DATA XREF: seg000:0091E87C↓o
seg000:0000034D                 dec     ecx
seg000:0000034E                 mov     ch, 1Ch
seg000:00000351                 mov     ch, 36h ; '6'
seg000:00000353                 push    ecx
seg000:00000354
seg000:00000354 loc_354:                                ; DATA XREF: seg000:0090F7F4↓o
```

Figure 4: An execution trace showing context switching in the virtualized program

4.2 The VM scheduling process

In this paper dynamic scheduling scheme generated by the two control units:
The segment control unit randomly decided that control should be given to the dispatcher or bytecode handler.
Switching VM Unit, random selection of VM for use.

Dynamic scheduling present into an algorithm, the instruction of bytecode will be executed sequential mode. The virtual interpreter fetches standard bytecode to dispatches to the handler. After completion of bytecode execution, the control unit decision, program control goes back to Dispatcher or VM handler if the control goes back to bytecode dispatcher, Dispatcher, and VM random select executed bytecode from standard bytecode. If the control moves to the Bytecode handler, execution will be held from offset Bytecode. This process continues till All virtual instruction of the protected code region will not end.

5. Case Study

5.1 Defense Process

To demonstrate how our approach protects the APK (Android Application Package), we have chosen a game app 2048.apk [23] to describe the defense process.
Step1: As given in Fig. 4, first of all, try to obtain binary SO file from unzipping app packages; secondly, disassemble the app to get the key-code section. We undertake the offset start from 00x328 and a close offset address 00x354. Step 2: Segment of instructions being protected are virtualized one by one. As depicted in Fig. 5, select the random set of virtual instruction into VMData for binary bytecode presentation.

Step 3: Design the relating VM as per the custom configuration decision. Showing in Fig. 5, if the configuration decision is VM02, it will naturally deliver a practically equipollent usage for a particular handler. Every VM has various handles execution arrangements for each running, and the planning tables between virtual directions and the overseers are additionally profoundly influential. In particular, the control flow goes through interminable changes

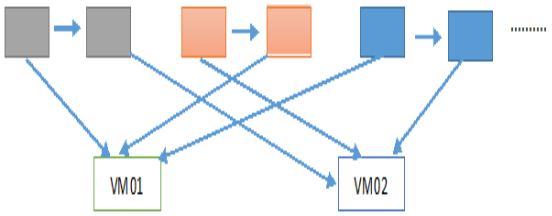

Figure 5: Represent handlers entrenched in VM under tradition implementation, and similar stripe boxes denote the functionality of handler

Step 4: The critical thing under this step to monitor the VM jumping entry point; Garbage instruction is associated with the implementation of obfuscation due to the VM jumping statement.

Step 5: In the design system, a new code segment (VMSection) embed into the SO file; VMSection is a very composed method of multiple VM, VMData, and VMInit, etc.
Step 6: In the last advance, the framework repackages the beginning application. The capacity of the form early on incited paper newly_2048.apk is indistinguishably identical to the pre-sponsorship one.

5.2 Running process

In this section, we represent how we protect our app in the form of execution, and we illustrate some technical detail here about the New2048.apk
Step 1: In this step, new-2048.apk execute to the protected code segment, Fig. 3, showing that protected apk jump on VMInit. The VMInit initializes the virtual machine. All protected instructions perform to achieve the correct

restoration; VMContex saves the real-time environment register value for simulating the CPU register behavior.

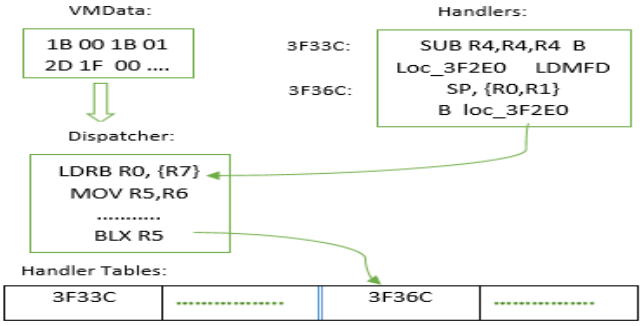

Figure 6: Process Model of Dispatcher

Step 2: An important step is to initialize the VMContex, Inside the VMData, Bytecode executed by Dispatcher. Our approach chooses the equivalent set of VM to perform as per the guideline of virtual Instructions. In Fig. 6, In this paper, we select VM01 for illustration, begin the decision to choose the VM, select the equivalent handler to execute to obtain the offset. We demonstrate an example of the parsing process based on Fig. 6; VMData keeps the address of register R7. The program controller starts executing the handler from offset 0x3F36C.

Step 3: VM interpreter chooses the first handler to execute; here, one genuine instruction is planned semantically into multiple virtual ones. When this process successfully over dispatcher jump to entrance offset 0x3F2E0.

Step 4: Program Bytecode execution inside the VMData; once complete bytecodes fetched out, the program controller jumps to Step 5; otherwise, it reiterates Step 2.

Step 5: All VMData bytecodes are interpreting and executing, then the program moves to HD_Exit; VMContext saved the latest value of real registers.

Step 6: Finally, the controller moves the forfended code segment endpoint address and continuous, executing outside VM instruction.

6. Evolution:

It provides the protection of binary protection of file, attacker unable to make reverse engineering attacks. We evaluate the performance of our approach by using a different type of attacks on protected apps.VM based MUltiDEX protection approach using Multi VM technique to safeguard the SO and MultiDex files. To verify the protection level of our system, have the two type of analysis

I. Static analysis
II. Dynamic analysis

6.1 Static analysis:

Many Static tools are available in the market to launch the reverse engineering attack to repackage the app. We found JEB [24] and Apktool [25] can disassemble the parse DEX and MultiDEX files system for reverse engineering.

Similarly, we have IDA-Pro, which can parse the SO file into ARM Instruction. One more essential approach of potential vulnerability JNI-OnLoad by the cracker. This JNI-OnLoad function is used in IDA-Pro for debugging.

6.2 Dynamic Analysis:

We found six shelling tools to test the different approaches. We select the three latest tools to test our system. DexHunter[6] uses this tool to unpack the DEX file from the app, restore all instruction, and modifies the code. As above shown in the table, DextHunter unable to unpacking the VM Based MultiDEX protection files. PackerGrind [8] can analyze the DEX Files, loading class, and unpacking the app at runtime. The significant advantage of this app is to monitor the memory operation at runtimes. App protected from approach, have no memory operation VM base MultiDEX protection app safe from packaging tool. DroidUnpack [26]is a potent tool for reverse engineering; it works on a multi-layer detection environment. It has hidden code extraction, self-modified code.

7. Result Analysis

We compare our approach from other previous systems; the present system protects only one DEX class file, in the current scenario, with rapid development in-app size. Every APK has the MultiDEX file, shown in Fig. 7; our approach protected MultiDEX files and SO files. In this segment, we analyze the VM-based MultiDEX protection in terms of space and time complexity. Time complexity calculates the apps startup time and space complexity calculated by the app size and consuming memory by the app at run time. We applied our VM Based MutltiDEX approach to protecting the binary SO file on the priority in the first section. In the second section, we select the 16 apps category under the 1,000,000 popular download apps from google play. Detail of selected apps given in Tab. 3. Previous works shown DSVMP [27] suggest 5 VM Configuration is the latest for virtual machine protection. Therefore, we select the 5 VM configuration overhead for experimental results, suitable for modern virtual machine protection. The key factor in judging app protection is size; if any app protection size is large, it is better protection. Still, our approach uses minimal space for protection in multi VM environments. In Fig. 8, we found that the protected app increased by 15.81%. If we evaluate the app 2048.apk, this app

increased by 17 % in protection apps; some apps increased widely in Virtual Machine protection. An increase in Protection app size depends on VMSection, some apps have an initial size of the app, and growth in VMSection is a very close or maybe very high rate in the development of VMSection. Next, we measure the VM Based MutltiDEX approach to provide the average 15.81 % increase in protection rate, shown in Fig 8. It is an adequate level for any algorithm level program. We found startup time overhead of from 1.23% to 6.83%.

Due to various approaches, we compared two commercial approaches for the protection of MultiDEX and SO files. We selected the UPX- Shell [9] and Hikari[28] approach for comparison. Hikari gave the very worst performance in comparison to UPX-Shell and our VM-based MultiDEX approach. UPX-Shell gave very close results from our VM base MultiDEX approach, but our system doesn't have a decryption and unpacking process at the initial stage. In conclusion, our system has time complexity, and space complexity is more effective than UPX-shell., showing in Tab. 5 and Fig. 8

Table 3: Different type of APK with MultiDEX file and size of APK

Name of APK	MultDEX Files	Size of APK (MB)
2048_v4.6.2.com.apk	3	16.5
TikTok_v17.8.4.com.apk	8	68.2
Snapchat_v11.4.5.73.com.apk	4	69.8
Netflix_v7.80.0 build 10 35175.com.apk	4	58.3
Viber Messenger Group Chats_v14.1.0.16.com.apk	4	53.8
Shan Koe Mee SKM777_v1.1.com.apk	2	75.1
Golden Card Games Tarneeb Trix Solitaire_v20.1.0.27.com.apk	1	12.3
VideoBuddy YouTube Downloader_v1.36.136000.com.apk	4	16.7
Waze GPS Maps Traffic Alerts Live Navigation_v4.68.1.0.com.apk	4	85.1
WATCHED Multimedia Browser_v1.0.2.com.apk	1	21.6
NVIDIA GeForce NOW_v5.33.29272392.com.apk	2	43.3
Yahoo Mail – Organized Email_v6.14.1.com.apk	4	25.9
KineMaster Video Editor Video Maker_v4.15.9.17782.GP.com.apk	3	90.9
MX Player_v1.24.6.com.apk	3	33.2
TextNow Free Texting Calling App_v20.42.0.2.com.apk	4	80.6
Discord Talk Video Chat Hang Out with Friends_v46.3.com.apk	2	76.9
Average	3.33	54.11

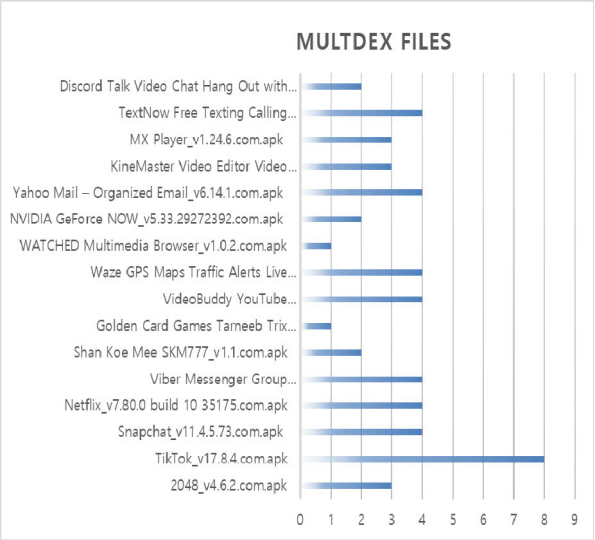

Figure 7: Different APKs with MultiDEX Files

Table 4: Represent the APK volume, Memory consumption and startup time of VM based MultiDEX protection

VM Based MultiDEX Appraoch			
Name of APK	**APK Volume (MB)**	**Memory Used(MB)**	**Startup time(MS)**
2048_v4.6.2.com.apk (16.5 MB)	19.8	73.8	882.7
TikTok_v17.8.4.com.apk (68.2 MB)	95.5	304.9	3648.5
Snapchat_v11.4.5.73.com.apk (69.8 MB)	77.8	310.7	3718.1
Netflix_v7.80.0 build 10 35175.com.apk (58.3 MB)	67.0	260.6	3118.9
Viber Messenger Free Video Calls Group Chats_v14.1.0.16.com.apk (53.8 MB)	63.4	238.3	2851.4
Shan Koe Mee SKM777_v1.1.com.apk (75.1 MB)	101.9	337.5	4039.0
Golden Card Games Tarneeb Trix Solitaire_v20.1.0.27.com.apk (12.3 MB)	19.6	73.8	882.7
VideoBuddy YouTube Downloader_v1.36.136000.com.apk (16.7 MB)	22.5	74.7	893.4
Waze GPS Maps Traffic Alerts Live Navigation_v4.68.1.0.com.apk (85.1 MB)	100.4	380.4	4552.6
WATCHED Multimedia Browser_v1.0.2.com.apk (21.6 MB)	28.3	96.6	1155.5
NVIDIA GeForce NOW_v5.33.29272392.com.apk (43.3 MB)	57.2	193.6	2316.4
Yahoo Mail – Organized Email_v6.14.1.com.apk (25.9 MB)	32.0	115.8	1385.6
KineMaster Video Editor Video Maker_v4.15.9.17782.GP.com.apk (90.9 MB)	106.4	406.4	4862.9
MX Player_v1.24.6.com.apk (33.2 MB)	43.2	148.4	1776.1
TextNow Free Texting Calling App_v20.42.0.2.com.apk (80.6 MB)	92.7	360.3	4311.9
Discord Talk Video Chat Hang Out with Friends_v46.3.com.apk (76.9 MB)	93.0	343.8	4113.9
Average	63.80	232.47	2703.92

Table 5: Comparison of different previous techniques after the protection of 16 Apps

Performance	Initial size	MultiDex VM	UPX Shell	Hikari
APK Volume	52	63.80	47.38	78.00
Memory Used	197.6	232.47	219.93	302.71
Startup time(ms)	2597.92	2781.85	3637.09	3637.09

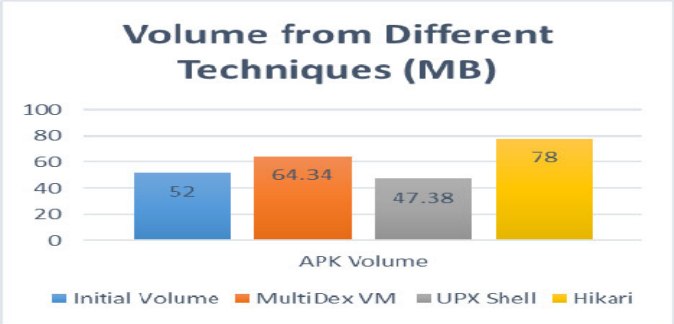

Figure 8(a). The volume of APK

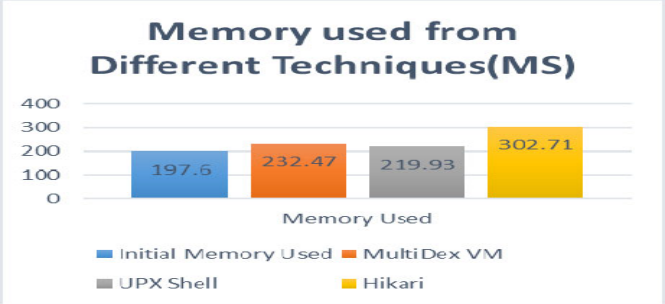

Figure 8(b). Memory Consumption

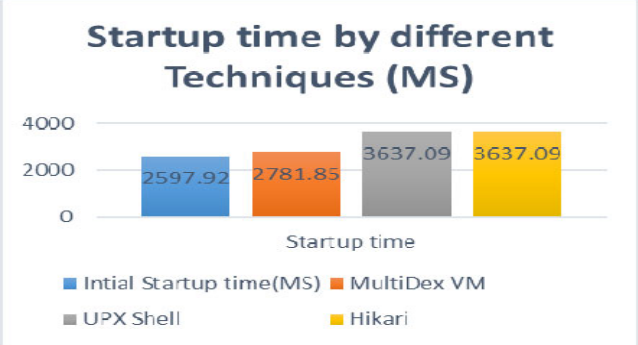

Figure 8(c). StartUp time

Figure 8: Average Performance of different techniques after the protection of 16 Apps

8. Conclusion and Future works

This paper introduces a VM-based Multidex protection approach; we used the Stack-based native system in this approach. It provides intense level protection under the virtual machine. The significant advantage of this approach to protect the MutiDex files and SO files is that very few systems are available, which take care of both types of files. In this approach have both static and dynamic observation in the experiment. Our system is shown as a result of low overhead and better performance in time and space complexity. We observe that the UPX shell using less memory consumption than VM based MultiDEX approach because we are using Multiple VM with diversity. In the future, we will try to reduce the memory consumption of than UPX shell.

9. References:

[1].https://arstechnica.com/information-technology/2020/03/found-malicious-google-play-apps-with-1-7-million-downloads-many-by-children/.

[2]. Luo, L., Fu, Y., Wu, D., Zhu, S., & Liu, P. (2016, June). Repackage-proofing android apps. In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 550-561). IEEE.

[3]. Vidas, T., Votipka, D., & Christin, N. (2011, August). All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In Woot (pp. 81-90).

[4] (2015). Dexextractor. [Online]. Available: https://github.com/lambdalang/DexExtractor

[5] Jack Jia. (2014). Android App Dynamic Reverse Tool Based on Xposed Framework. [Online]. Available: https://github.com/halfkiss/ZjDroid

[6] Zhang, Y., Luo, X., & Yin, H. (2015, September). Dexhunter: toward extracting hidden code from packed android applications. In European Symposium on Research in Computer Security (pp. 293-311). Springer, Cham.

[7] Zhou, W., Wang, Z., Zhou, Y., & Jiang, X. (2014, March). 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).

[8] Xue, L., Luo, X., Yu, L., Wang, S., & Wu, D. (2017, May). Adaptive unpacking of Android apps. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 358-369). IEEE.

[9] Fish_Ou.(2015). Android SO UPX. [Online]. Available: https://www. cnblogs.com/fishou/p/ 4202061. html

[10] GitHub. (2014). UPX Shell Tools. [Online]. Available: https://upxshell.en.softonic.com/
[11] Junod, P., Rinaldini, J., Wehrli, J., & Michielin, J. (2015, May). Obfuscator-LLVM--software protection for the masses. In 2015 IEEE/ACM 1st International Workshop on Software Protection (pp. 3-9). IEEE.
[12] Currwin. (2015). DecLLVM. [Online]. Available: https://github. com/F8LEFT/DecLLVM
[13] (2012). Proguard. [Online]. Available: https://www.guardsquare.com/en/ products/proguard
[14] (2017). Dexguard. [Online]. Available: http://www.saikoa.com/dexguard
[15] Fang, H., Wu, Y., Wang, S., & Huang, Y. (2011, October). Multi-stage binary code obfuscation using improved virtual machine. In International Conference on Information Security (pp. 168-181). Springer, Berlin, Heidelberg.
[16] Cohen, R., & Wang, T. (2014). Android Application Development for the Intel Platform (p. 520). Springer Nature.
[17] Kim, N. Y., Shim, J., Cho, S. J., Park, M., & Han, S. (2016). Android Application Protection against Static Reverse Engineering based on Multidexing. J. Internet Serv. Inf. Secur., 6(4), 54-64.
[18]. Lim, K., Kim, N. Y., Jeong, Y., Cho, S. J., Han, S., & Park, M. (2019). Protecting Android Applications with Multiple DEX Files Against Static Reverse Engineering tacks. INTELLIGENT AUTOMATION AND SOFT COMPUTING, 25(1), 143-154.
[19] He, Z., Ye, G., Yuan, L., Tang, Z., Wang, X., Ren, J., ... & Wang, Z. (2019). Exploiting Binary-Level Code Virtualization to Protect Android Applications Against App Repackaging. IEEE Access, 7, 115062-115074.
[20] Hex-Rays. (2015). IDA Protect. [Online]. Available: https://www.hexrays.com/index.shtml
[21] Kuang, K., Tang, Z., Gong, X., Fang, D., Chen, X., & Wang, Z. (2018). Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. Computers & Security, 74, 202-220
[22] J. Seward. (2013). Valgrind. [Online]. Available: http://valgrind.org/
[23] Google Play. (2018). 2048.apk. [Online]. Available: https://play.google. com/store/apps/ details?id=com.androbaby.game2048
[24] PNF Software. (2015). JEB. [Online]. Available: https://www. pnfsoftware.com/
[25] R. Wisniewski. (2010). Apktool. [Online]. Available: https://ibotpeaches. github.io/Apktool/
[26] Duan, Y., Zhang, M., Bhaskar, A. V., Yin, H., Pan, X., Li, T., ... & Wang, X. (2018, February). Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation. In NDSS.
[27] Kuang, K., Tang, Z., Gong, X., Fang, D., Chen, X., & Wang, Z. (2018). Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. Computers & Security, 74, 202-220.
[28] Tencent. (2017). Hikari. [Online]. Available: https://github. com/HikariObfuscator/Hikari

M A Rahim Khan received the M.Tech. Degrees in Information Technology from GGIP University New Delhi in 2008, he worked as a lecturer at Majmaah University, Saudi Arabia. Currently Ph.D. Scholar in Lingayas Vidyapeeth Faridabad India. His research area in Cyber Security, Computer Network Security, Information Security.

Dr. Manoj Kumar Jain (Associate Professor CSE & In-charge Academics) Lingayas Vidyapeeth Faridabad India. His research area is Neural Network, Information Security, Wireless Sensor Network. He has published several research papers in Nation and international journals and organized and participated in various national and international conferences.