
Article:
Zhao, Y, Tang, Z, Ye, G et al. (4 more authors) (2020) Compile-time Code Virtualization for Android Applications. Computers & Security, 94. 101821. ISSN 0167-4048

https://doi.org/10.1016/j.cose.2020.101821

Compile-time Code Virtualization for Android Applications

Yujie Zhao[1], Zhanyong Tang[1], Guixin Ye[1], Dongxu Peng[1], Dingyi Fang[1], Xiaojiang Chen[1], Zheng Wang[2]

***Abstract*—Infringing intellectual property by reverse analysis is a severe threat to Android applications. By replacing the program instructions with virtual instructions that an adversary is unfamiliar with, code obfuscation based on virtualization is a promising way of protecting Android applications against reverse engineering. However, the current code virtualization approaches for Android only target at the DEX bytecode level. The DEX file with the open file format and more semantic information makes the decode-dispatch pattern easier to expose, which has been identified as a severe vulnerability of security and can be exploited by various attacks. Further, decode-dispatch interpretation frequently uses indirect branches in this structure to introduce extra overhead. This paper presents a novel approach to transfer code virtualization from DEX level to native level, which possesses strong security strength and good stealth, with only modest cost. Our approach contains two components: pre-compilation and compile-time virtualization. Pre-compilation is designed for performance improvement by identifying and decompiling the critical functions which consume a significant fraction of execution time. Compile-time virtualization builds upon the widely used LLVM compiler framework. It automatically translates the DEX bytecode into the common LLVM intermediate representations where a unified code virtualization pass can be applied for DEX code. We have implemented a working prototype Dex2VM of our technique and applied it to eight representative Android applications. Our experimental results show that the proposed approach can effectively protect the target code against a state-of-the-art code reverse engineering tool that is specifically designed for code virtualization, and it achieves good stealth with only modest cost.**

***Index Terms*—Android packer, code visualization, compiler, LLVM.**

I. Introduction

With open-source and free, Android has achieved a dominant position in the mobile device market. The market share of Android in the worldwide smartphone market has reached 87%, according to IDC study [4]. However, in Android ecosystem, the fact that Java class files and DEX files contain much semantic information makes it exceptionally difficult to protect the underlying source. And yet most software developers continue to ignore the consequences, leaving their intellectual property at risk.

To address this problem, most of the prior work relies on a packer to strengthen Android applications. The underlying principle of the packer is to embed a piece of code in the binary program to obtain priority control of the program and hide the execution logic of the application through code obfuscation [13], [56], encryption [3], and other technologies, while ensuring the correctness of its running results. The Android packer is more complicated with respect to both DEX and native levels since the Android system has a multi-level design with runtime Dalvik(version 4.4 and earlier) and ART (version 5.0 and later). Currently, the state-of-the-art Android packer technique is to incorporate with code virtualization at the Java layer [68], [46].

However, code virtualization at the Java layer has some limitations. 1) Vulnerability. Like most code virtualization implementations, the original DEX file adds a new section with more content, including a virtual instruction set, a virtual instruction scheduler with a decode-dispatch pattern, which has been identified as a severe vulnerability of security and can be exploited by various attacks [59], [63], [15], [45]. The open and semantic DEX file format makes it easier to expose these sensitive virtualized information. Because many existing DEX analysis tools make it easy to obtain the $DexFile$ data structure. Such information can assist the accurate collection of control flow and data flow of the virtualized code. Even if the static analysis of virtualized code may not reveal the complete semantics of a packed app, the behavior of the decode-dispatch pattern should be exposed to the attacker. 2) Performance. A recent study [59] shows that the slowdown varies from 1.9X to 660.9X when only 10% of the code is virtualized. The performance issue is even more urgent on mobile devices like smartphones when implying code virtualization on the Android app.

We propose a novel and reliable Android app packer called compile-time virtualization, which possesses strong security strength and good stealth, with only modest cost. Our key insights have two points: 1) Native code is more difficult than DEX code in terms of readability and simplicity as it preservers less semantic meaning of the program. For example, native code contains no symbol information, so that variables are mapped to registers and many symbols are just an address [58]. Native code almost with no semantic information is hard enough to analyze, and it will even harder if they are translated into customer-defined instructions which attackers are not familiar with. 2) Native code is 2 to 5 times faster than the original DEX bytecode according to Googles experiments on the mobile device Nexus One [57]. If part of the DEX bytecode is converted to native code, the performance improvement brought by it can offset part of the overhead introduced by code virtualization.

So, the main idea of our scheme contains two components of pre-compilation and compile-time virtualization. Pre-compilation is designed for performance improvement and compile-time virtualization overcomes the vulnerability of

1 Authors are with the School of Computer Science and Technology, Northwest University, China.

2 Zheng Wang is with University of Leeds, United Kingdom.

security. Pre-compilation is to identify and decompile the critical functions which consume a significant fraction of execution time into C/C++ codes. Compile-time virtualization is built on the LLVM(Low Level Virtual Machine) [35]. First, its front end uses GCC compiler to translate C/C++ code into LLVMIR. Second, it replaces its middle representation LLVMIRs(Intermediate Representation of LLVM) with custom instructions, and finally its back end compiles virtualized LLVMIR into native code.

Unlike past approaches, we use the idea of pre-compilation in a way like AOT(Ahead Of Time) compiler. This kind of method takes advantage of the natural properties that Android supports NDK development [23], solving the long-standing performance issue introduced by app packers. Further, we also propose a double-layer Android packer to transfer code virtualization from DEX level to native level.

There are some challenges in our work. First of all, completely transforming the whole DEX file into native code is such a challenging task that will lead to excessive overhead. We propose a Decision-making model to determine which function in a DEX file is worth pre-compiling in terms of the execution time. Secondary, how to implement pre-compilation directly on DEX bytecode without Java source code. We implement a decompilation engine that converts DEX bytecode into C/C++ code. Finally, how to make the decode-dispatch pattern more hidden and not easy to be found by attackers. We propose a hiding method to change the control flow of the decode-dispatch, then take advantage of passes in the LLVM compiler framework, compiling the virtualization related information into a new SO file. Although the newly generated SO file still contains the decode-dispatch pattern, compared to the protected DEX file, it exposes less semantic and structural information to the attacker.

Summary of Results:

We develop a compile-time code virtualization tool of Dex2VM based on LLVM and evaluate it with four indicators of resilience, stealth, cost, functionality. They represent the ability to resist real reverse analysis or deobfuscation tools, the existence of obfuscation, performance overhead, and functional integrity. In the experiment, the Android versions are 4.4.4r1(with Dalvik runtime), 5.0(with ART runtime), and 9.0(the latest version). Our experiments lead to the following results:

- Our experimental cases of resilience include manual attack, general unpacker tool attack, and the-state-of-art deobfuscation on virtualization, but all fail on Dex2VM. The experimental results demonstrate that Dex2VM is effective for code protection.
- As to stealth, we use Artificiality [29] and sample register address space. Compared with other well-known protection tools of Android applications, such as Tigress [14] and OLLVM [26], the experimental results show that Dex2VM is still the least exposure of sensitive features.
- Our performance test cases include CPU, size, memory overhead, power consumption, and 0xbench suite [1]. Compared with the original app, the experimental results show that when 20% of the code is virtualized, the power consumption decreased by 6%, and memory overhead increased by more than 5%.
- According to user study results, compared with the original app, the protected one by Dex2VM has no significant changes except for the increase in installation time.

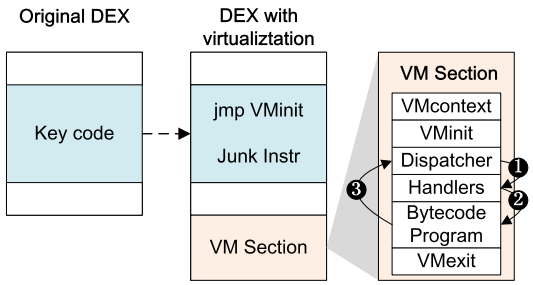


Fig. 1. The classic code virtualization on DEX. (Decode-dispatch based interpretation: Step 1–3 forms a central loop to dispatch, decode and execute the bytecode. Decode-dispatch pattern has been identified as a severe vulnerability of security and can be exploited by various attacks. The open file format of DEX makes it easier to expose.)

Contributions: This paper makes the following contributions:

- It presents the first reliable scheme that transfers code virtualization from DEX level to native level.
- It is also the first to apply virtualization at the compilation phased for Android app protection, presenting a language-independent and platform-independent virtualization architecture. As a result, the design delivers all kinds of Android system resolution.
- It is the first to propose a solution to improve performance overhead when doing virtualization on the android app.
- It does experiments from dimensions of resilience, stealth, cost, and functionality to verify the effectiveness and feasibility of our approach.

II. Background and Motivation

In this section, we describe the motivation and attack model. In addition, we introduce JNI and LLVM, with which we put our idea into practice.

A. Motivation

Code virtualization converts the critical code segment of the original DEX bytecode to written in a custom-defined virtual instruction set, which the attacker is not familiar with. A new VM section that contains all code virtualization related information will be linked (or inserted) into the original DEX file, where the entry point of the protected code region will be redirected to a function call to invoke the VM to translate the bytecode instructions.

Figure 1 illustrates the classic way to implement code virtualization on DEX bytecode with the decode-dispatch pattern. A dispatcher that determines which instruction is ready for execution, a set of bytecode handlers that first decode the bytecode and then translate it into native machine code. Step 1-3 forms a central loop to dispatch, decode, and execute the bytecode. Many previous works perform reverse analysis to identify these central loops and then find the mapping between each bytecode and its corresponding handler

function [59], [63], [15], [45]. Further, the open file format of DEX provides enough semantic information for attackers to discover new sections. With many existing DEX analysis tools, such as DexHunter [66], TIRO [58], PackerGrind [61], DROIDUNPACK [18] etc., attackers can quickly find the $DexFile$ data structure, and then obtain the control flow and data flow of the program, which makes the decode-dispatch pattern easier to expose.

In addition, decode-dispatch interpretation frequently uses indirect branches in this structure to introduce extra overhead. That is why code virtualization always results in increased performance overhead [19]. However, Android usually runs on a variety of smartphone devices aiming at the requirement of high-performance, including fast speed, small size, and low power consumption. Therefore, it is a huge challenge to implement code virtualization on Android while overcoming the limitations of performance. This motivates us to design compile-time virtualization for Android.

B. The Attack Model

The classical approach to reverse analysis on a VM-protected program typically follows three steps [33]:

$Step1$: Find the entry point address of the VM interpreter.

$Step2$: Find the mapping between each bytecode and its corresponding handler function.

$Step3$: Recover the logic of the target code region.

Our attack model assumes that the attacker has accumulated some experience and skills in reverse analysis and is familiar with the environment of Anroid ecosystem. We also assume that the attacker can skillfully use reverse analysis tools such as IDA [42], Ollydbg [65], etc. Further, we assume that the attacker has the ability to perform static analysis and dynamic analysis, and is capable of debugging, tracing, and even modifying memory. The aim of the attacker is to completely reverse the internal implementation of the target program. Our goal is to increase the difficulties in terms of time and efforts for an attacker to reverse the app protected using VM-based code obfuscation.

C. JNI

JNI(Java native interface) acts as a bridge between the Java code and native code, allowing them to interact with each other. JNI functions are declared within Java level but defined in native libraries. There are mainly two ways of registering the native code in JNI. One is static registration of naming the JNI functions in a specific way like $Java$ + $package\_name$ + $class\_name$ + $function\_name$ format so that the function mapping is automatically handled. The other is dynamic registration of mapping the native code and Jave code through $RegisterNatives()$ method in $JNI\_Onload()$.

In the pre-compilation of Dex2VM, JNI is used to sink code from DEX level to native level. From the app analysis point of view, JNI breaks the control flow and data flow analyses so that we can leverage JNI to hide sensitive behaviors from being detected. Further, some functions with heavy computations can be written as native but still be called from Java level to improve performance.

D. LLVM

As the most popular design for a traditional static compiler, LLVM [38], [67] compiler infrastructure is also a Three-Phase compiler contains three major components, the front end, the optimizer, and the back end. The front end is responsible for parsing, validating, and diagnosing errors in the input code, then translating the parsed code into LLVMIR. The optimizer does a series of transformations to try to improve the IR's running time. The back end is to produce native machine code. LLVM Intermediate Representation (IR) is the form LLVM uses to represent code in the compiler. Any language can be converted into IR, and the same IR can be converted into any architecture assembly language.

In this paper, we replace the LLVMIR instruction with a custom-defined virtualization instruction, which can effectively improve security. Because depend on LLVM, the back end compiles the virtualized LLVMIR and related information into a new SO file. Although the newly generated SO file still contains the decode-dispatch pattern, compared to the protected DEX file, it exposes less semantic and structural information to the attacker.

III. Overview

In this section, we introduce the workflow of Dex2VM and describe the excution flow of a protected function.

A. WorkFlow

Dex2VM is a compile-time code virtualization tool that has essentially put our idea into practice. The tool takes in a DEX file to be protected. It produces virtualized native codes in a new SO file. Dex2VM also modifies the original DEX bytecode by replacing the original critical functions bytecode with a native header by JNI. Figure 2 depicts the five steps of building a Dex2VM using our approach. Each of the steps is described as follows.

Step 1. Extracting functions. In this step, the Decision-making Model is used to identify critical functions in the DEX file to be protected. It calculates each function's occupancy in terms of execution time and adds the functions whose occupancy exceeds a predefined threshold to the whitelist. This is detailed in Section IV-A1. The DEX binaries of functions on whitelist are disassembled into a human-readable Smali codes by using baksmali [25] and dexdump tool.

Step 2. Decompilation Engine. Decompilation Engine takes in Smali codes and whitelist. On the one hand, it translates Smali codes into C/C++ codes by the decompilation engine described in Section IV-A2. On the other hand, it registers all functions on the whitelist in DVM(Dalvik Virtual Machine) by using JNI. The output of this step is a modified DEX file by replacing the original critical function's bytecode with a native header, and a new C/C++ file.

Step 3. Clang. C/C++ code is compiled by Clang that is the front-end of LLVM to generate LLVMIR. C/C++ code is an intermediate language for translating DEX bytecode to LLVMIR in Dex2VM.

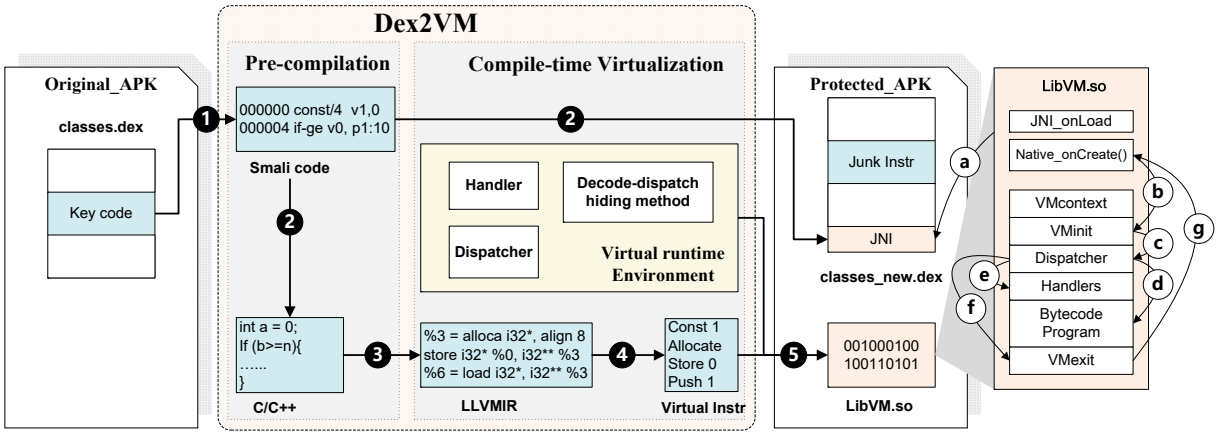


Fig. 2. Overview of Dex2VM. The workflow is shown as the following steps: ① Some critical functions is selected by Decision-making Model to pre-compilation. ② A decompilation engine is to convert DEX bytecodes to C/C++ codes. ③ C/C++ codes are compiled by front-end Clang to generate LLVMIRs. ④ LLVMIRs are replaced by the custom-defined instructions of Dex2VM. ⑤ A protected apk with new DEX and a virtualized SO is produced. Runtime execution flow is shown as the following: ⓐ Register JNI. ⓑ Jump into VM. ⓒ Initialize VM and enter Dispatcher. ⓓRead virtual instruction bytecode. ⓔ Dispatch handlers to process bytecodes. ⓕ Exit VM. ⓖ Go back the instruction sequence and continue.

Step 4. LLVMIR virtualization. By using LLVM as a compilation framework, Dex2VM defines its own virtual instructions in Section IV-B1, handler and dispatcher in Section IV-B2. To further reduce the exposure of decode-dispatch pattern, a hidden method of dispatching mechanism is introduced in Section IV-B3.

Step 5. Apk repackaged. At the end, the package with the modified DEX bytecode and the Dex2VM protected native code is built into a new application(.apk). Finally, the newly protected application can be executed on the DVM and can switch between the native side and DVM with the help of JNI.

B. Execution Flow

We take $onCreate()$ as an example to describe the execution flow of the protected function. After protection of Dex2VM, it is implemented in the native level and named with $native\_onCreate()$. It is eventually compiled into a $LibVM.so$ file. When the application starts, $LibVM.so$ is loaded into the current application process, and completes the dynamic registration of $native\_onCreate()$ in $JNI\_Onload()$.

When the application triggering $onCreate()$, DVM transfers string of $native\_onCreate()$ to a native process by JNI. The natvie process passes $Native\_onCreate()$ signature to the Dex2VM interpreter, which locates the starting position of the virtual instruction for $native\_onCreate()$. The VM in $LibVM.so$(hereinafter referred to as LibVM) finds the corresponding location to complete the initialization.

The LibVM reads the virtual bytecode and enter into a central loop to dispatch, decode, and execute the bytecode. During the virtual execution process, when encountering the function defined in Java level, the LibVM splices out Java method names and parameters through virtual instructions, then passes this information to the related JNI functions, and transfers execution permissions to the DVM.

Until the execution of the Java layer function ends, the LibVM regains execution permissions. At this time, if other native layer functions need to be executed, the dispatcher saves the currently virtual instruction address in the virtual stack. Then LibVM exits and hands over the execution rights to other native process. When other native functions finished, the execution permission returns to LibVM. Then LibVM re-reads the virtual instruction address stored at the top of the stack and continues execution.

After all the virtual interpretations finished, LibVM transfers the operation results from the virtual stack to its caller and destroys the stack space and virtual register space. The $native\_onCreate()$ converts the obtained return value to the required type and passes it to DVM. Finally, the Java layer functions continue to be executed.

From this example, we can see that execution permissions are continuously passed among the DVM, local process, and LibVM, which mostly destroys the program's control flow and data flow to increase the difficulty of the attackers reverse analysis. Obviously, the frequent interaction between Java and native levels will also bring additional overhead. That is why we introduce a pre-compilation to reduce the performance overhead.

IV. Implementation

A. Pre-compilation

In this section, we will introduce the design thinking of Dex2C. The framework is composed of two components, a Decision-making Model and a decompilation engine. The Decision-making Model is used to find out which functions are suitable for decompilation. The decompilation engine is used to translate selected DEX bytecode into C/C++ code.

1) Decision-making Model: Transforming the entire DEX bytecode into native code is a big challenge. That's because JNI suffers from time and space overhead, just like other mechanisms of supporting interoperability. Further, even though a small vulnerability in native code would lead to a system crash. To trade off the balance of security and efficiency for the Android app, the goal of the Decision-making model is to select some critical functions and put them on the whitelist. The critical function here is defined as having

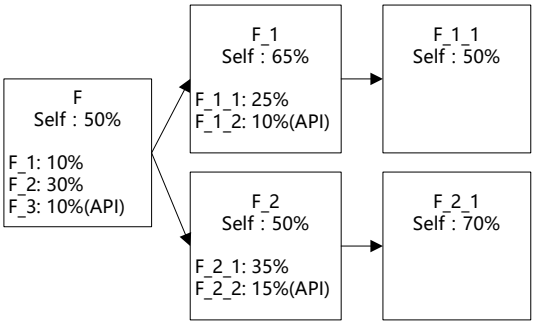


Fig. 3. An example of Decision-making Model strategies

two properties, one without recursion and the other with the ability to reduce function calls as much as possible.

The detecting mechanism of the Decision-making Model is to collect and count the occupancy of each function in the entire execution-flow and add the functions whose occupancy exceeds a predefined threshold to the whitelist. The function runtime ratio is collected via the CPU Profiler [22] provided by Google. The counting flow is organized into the following steps:

$Step1$: Recursively find all child functions of the current function, until there are no other function calls in the child functions, generating a function call tree.

$Step2$: The function call tree is traversed in post-order to determine which functions are pushed into the whitelist. If the current function's self-code execution rate (except for child functions and API functions) is over a pre-defined threshold, its name would be added to the whitelist. Mark its parent function as the selected function.

$Step3$: Check all child functions of the selected function, and superimpose the execution time rate of those child functions in the whitelist to the occupancy of the selected function.

$Step4$: If the total execution time rate of the selected function is over the pre-defined threshold, add its name on the whitelist.

An example is given in Figure 3 to show the occupancy calculation of a method. We assume that this is the function call tree generated after $step1$. Since the function F_1_1's self-code execution time rate is 50%, which is less than 60%, it cannot be added to the whitelist. Then we check its father function F_1, whose occupancy is 65% and is greater than 60%, so it is added into the whitelist. Next, when checking the father function of F_1, it is found that F also has the other child. So, along this branch, we find its leaf nodes F_2_1, whose occupancy is 70%. It means that F_2_1 is also pushed into the whitelist. Different from F_1, the occupancy of F_2 should be calculated by the occupancy of its child function. That is, the occupancy of F_2 is equal to 50% + (35% * 70%) = 74.5%, which is greater than the threshold of 60%, so F_2 is also in the whitelist. The final occupancy of F is 50% + 10% * 65% + 30% * (50% + 35% * 70%) = 78.85%, which indicates that the execution-flow started from function F spent 78.85% of its execution time on itself. Therefore, the final whitelist includes $\{F\_1, F\_2\_1, F\_2, F\}$

It is worth mentioning that Dex2VM users also can decide

```
add-int  v1,v0,p1
.....
const-string/jumbo   v1,0x00108
```

Fig. 4. Value passing instance in Dalvik virtual register. Register v1 is used to store int and string objects respectively.

which functions or classes are critical and in need of protection. These function and class names are added directly into the whitelist.

2) Decompilation Engine: There are two challenges to resolve when decompiling DEX bytecode into C/C++ code. They are type conversion and control flow construction.

Type Conversion: The bytecode instructions of DVM(Dalvik virtual machine) are register-based, and the number of the registers is up to 65536. All registers are 32-bit untyped. It means that all computations are handled at the register level by using almost unlimited numbers of virtual registers. For example, a 64-bit data is composed of two adjacent 32-bit registers. Therefore, it is a common phenomenon that different types of data store in the same register during the execution of a piece of code. As Figure 4 shown that, in the first line instruction, register v1 stores an int object. While at the last line instruction, register v1 stores a string object. When translating the pieces of these codes, it is difficult to determine what the real value is in the register v1.

Java and C/C++ are both strongly typed language, and all variables must declare their types in advance. In a function, each variable has its specific life cycle as well as type. When DVM compiles Java source code with the DX tool, the maximum number of registers used by the function has been filled into the fixed field of the DEX file. To recognize all variables and their types, we store variables as well as all the registers they used into a map. The map will be updated if a new variable appears as the output of an assignment statement when translating the instruction.

Sometimes, temporary variables are introduced in the translation process, which is independent of the registers. When dealing with this kind of conversion, a random variable is obtained from the variable pool named intermediate variable assignment, and the final assignment is associated with the register.

Control Flow Construction: To restore the adjacency between instructions, we use the Path Tree to represent the semantic logic of the Smali instructions. We analyze the destination address of the jump instruction in the instruction sequence, obtaining the predecessor, successor nodes, out-degree, and in-degree of each node, then generate the Path Tree of the Smali instructions. For example, Figure 6 shows the Path Tree corresponding to the instruction fragment in Figure 5. The depth-first traversal is used to translate the nodes on each path until all nodes are covered. During the traversal process, the accessible variables in the current scope are continuously passed, relying on the parent-child relationship between the nodes. Also, the relationship between this variable and its register is passed. Each node completes the translation of

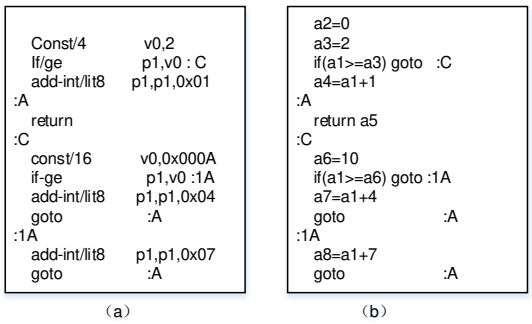


Fig. 5. Demonstration of the conversion process from the Smali instructions to the intermediate representations. (a) is the Smali instruction fragment,(b) is intermediate representations of C codes.

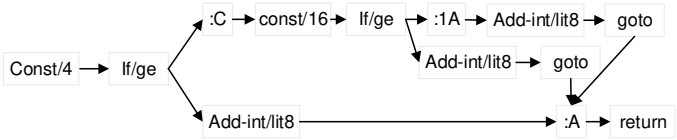


Fig. 6. Path Tree is used to scope and value transfer analysis for intermediate representations of C codes

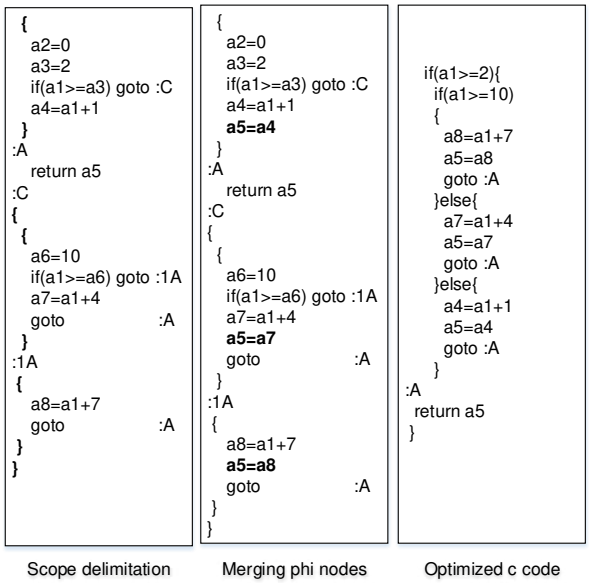


Fig. 7. Optimization and merging. The intermediate representations go through three steps to get the final C codes: Scope delimitation, Merging phi nodes, Optimized code.

the current node based on the available variables, generating intermediate representations of C codes.

The intermediate representations of C codes are not capable of execution, whose execution scope and value transfer should be analyzed based on the path tree. When traversing the Path Tree, starting from the node whose output is greater than 1, recursively search the first node whose degree of entry is greater than 1. Make the translation results of these nodes in the same scope until all nodes in their scopes. According to the corresponding relationship of variables between parent and child nodes, the value transfer dependency is established to merge phi nodes, so that the program has the correct execution process. The intermediate representation is optimized to delete irrelevant variables and dead code. As shown in Figure 7, a2 is not used after initialization, so it is an independent variable. A3 is only assigned once, and the assignment object is constant. Therefore, we deleted a2 and replaced a3 with the right value in the subsequent optimization process. We transform the optimized intermediate representations into C codes and write them into the corresponding C function body.

B. Compile-time Virtualization

In this section, we describe some critical techniques about the compile-time virtualization, which transforms the C/C++ code to native code by implementing code virtualization on LLVMIR.

1) Virtual instruction: The most critical aspect of LLVM's design is the LLVMIR [38], [67], which is the form it uses to represent code in the compiler. LLVMIR is defined as three address form, which means that it takes more registers. However, unlike most RISC instruction sets, LLVMIR is strongly typed with a simple type system. For example, i32 is a 32-bit integer, i32** is a pointer to pointer to a 32-bit integer. Another significant difference from machine code is that the LLVMIR uses an infinite set of temporaries named with a % character rather than a fixed set of named registers.

To increase the difficulty of reverse analysis, we define our instructions as no datatype. Therefore, it's a big challenge on how to use the typeless instructions simulating LLVMIRs without losing their semantics. We design a stack-based virtual instruction architecture similar to the JVM(Java Virtual Machine), even though its instructions have data types. The details of our custom-defined virtual instructions are shown in Table I. For example, when pushing the data of the corresponding type to the top of the stack, the JVM instruction set defines opcodes such as iconst, lconst, fconst, etc.. In this paper, the data push operation is defined as const virtual instructions, which requires the data to be aligned to the stack when it is stored, and converts to a specific data format when taken out. This not only simplifies the complexity of instruction design but also increases the complexity of code virtualization reverse analysis.

TABLE I
PARTIAL VIRTUAL INSTRUCTIONS AND THEIR FUNCTIONAL DESCRIPTIONS.

Virtual Instruction	Functional description
const	Define unformatted data and push it to the top of the stack
store	Put the data at the top of the stack in the specified virtual register, and allocate such space if the register represented by the index does not exist.
alloc	Dividing the memory area of the size specified by the top of the stack in the virtual stack space
load	Get unformatted data from the specified virtual register and push it to the top of the stack
icmp_eq	Compare the two data at the top of the stack and store the comparison result in the stack space (if it is equal, store unformatted data 1, otherwise save unformatted data 0)
goto	Change the control flow direction, jump to the specified virtual instruction to execute

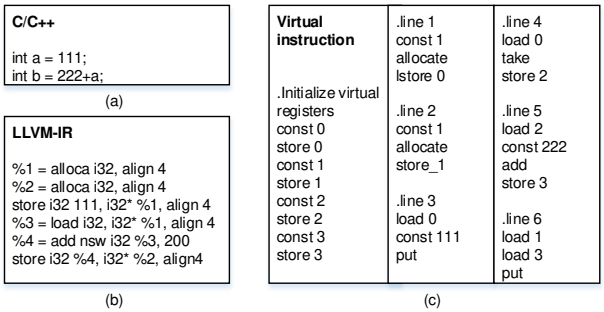


Fig. 8. Example native code snippet for a code region to be protected. (a) the original C/C++ code, (b) is LLVMIR, (c) is the virtual instructions.

LLVMIR is SSA(Static Single Assignment) based representation that means a variable must be defined before it used and be assigned only once. So, there is no doubt that the logic of the virtualized program is more complicated than the original one. As the Figure 8 shown, to express the same semantics, different languages need different numbers of instructions, C/C++ is 2, LLVMIR is 6, and virtual instructions are 27. The custom-defined instructions have three phases, which are virtual register initialization, virtual operation, and virtual register emptying. Firstly, according to the number and size of the temporary variables used in the LLVMIR, we use virtual instructions to simulate dynamic allocation registers in memory. Secondly, we design the virtual instruction to simulate the logic flow of the original program on the stack, where virtual registers are used as intermediate storage. Finally, all virtual registers' space is destroyed at the end of a virtual instructions segment.

2) Handler and Dispatcher: In a VM-based scheme, the execution path of the obfuscated code is controlled by a virtual instruction scheduler. A typical scheduler consists of two components: a set of bytecode handlers that translate bytecodes into native machine codes and a dispatcher that determines which bytecode is ready for execution. In this paper, according to the type of operation, virtual instructions are classified into three categories, and they are arithmetic operation, logical operation, and control flow transfer. The handlers are defined in Table II.

The dispatcher is used to simulate the execution process of a traditional CPU. There are actions such as fetching, decoding, virtual execution, and updating the address register. First, the dispatcher determines which bytecode is ready for execution. Then, it finds the corresponding handler to explain the current virtual instruction and decode it. Finally, after interpreting this instruction, the dispatcher continues to look for the next instruction to be executed.

3) Scheduling mechanism hiding method: In most cases, the dispatcher is the weakest point of a virtual machine to draw attackers' attention. To further increase the complexity of reverse analysis, a hidden method of dispatching mechanism is introduced in this paper. First of all, the control flow of the dispatcher is transformed into the if-else structure. Moreover, each basic block is split into two pieces with a flag as the boundary. Last, all the relationship between each small pieces of code components is shuffled to reduce the semantic information.

TABLE II
GENERATED HANDLERS FOR VIRTUAL INSTRUCTIONS.

Category	VI	Handler
Arithmetic Operation	and_i32	value v1=stack[stack_index- -] value v2=stack[stack_index- -] value v3=cast_i32(v1)&cast_i32(v2) stack[++stack_index]=cast_i64(v3)
	plus_i32	value v1=stack[stack_index- -] value v2=stack[stack_index- -] value v3=cast_i32(v1)+cast_i32(v2) stack[++stack_index]=cast_i64(v3)
Logical Operation	const	value v1=vmdata[vpc ++] stack[++stack_index]=cast_i64(v1)

	store	value v1=vmdata[vpc++];
		value v2=stack[stack_index- -];
		if(!Reg[v1]) alloc(Reg[v1]);
		Reg[v1]=cast_i64(v2)

Control Flow Transfer	ifne	value v1=vmdata[vpc++];
		value v2=stack[stack_index–];
		if(v2==0)
		vpc=find(Decrypt(vpc++),v1);
		else vpc=find(Decrypt(vpc+2),v1);

	goto	value v1=vmdata[vpc++];
		value v2=stack[stack_index–];
		find(Decrypt(vpc+2),v1);

Algorithm 1 is the pseudo-code of the dispatcher component hiding algorithm. First of all, find all the basic blocks in each module of the protected source code and save the address of the basic block in the global address table. Further, traverse all the instructions in the function, and judge whether the current instruction is a direct jump or a conditional jump according to the number of subsequent blocks of the instruction. Next, if it is a conditional jump, extract the condition of the jump, and put all subsequent block addresses into the private address table. An indirect jump is constructed with the private address table and the jump condition. If it is a direct jump, construct an instruction to read a destination address from the jump table and an indirect jump instruction. Finally, replace the original conditional jump or direct jump with an indirect jump.

Algorithm 1 Dispatcher Component Hiding Algorithm

Input: $Module$
Output: $Branch$

```
for Function in Moudle do
    for BasicBlock in Function do
        Branch.push(address_of_BasicBlock);
for instruction in BasicBlock do
    if instruction = branchInstruction then then
        if instruction_Successor >2 then then
            condition=getCondition(instruction);
            indrectBranch.push(Address(Successor0);
            Address(Successor1);
            branch=newBranch(condition,indrectBranch);
        else
            branch=newBranch(Branch(index))
        replace(branch,instruction)
return Branch
```

V. EVALUATION

In this paper, we evaluate the obfuscation in four aspects: resilience, stealth, cost, functionality. They are defined as the

following:

- **Resilience**: It measures the ability to withstand attack from an automatic deobfuscator.
- **Stealth**: It indicates how much more difficult in detecting the existence of obfuscation.
- **Cost**: It measures the execution overhead imposed by obfuscation.
- **Functionality**: It measures whether the function of app changed after obfuscation.

We evaluate Dex2VM and observe to what extent it meets these four criteria.

A. Environment Setup

The choice of experimental samples should follow the principle of universality, which means that the selected apps should cover a wide range of real-world apps. We apply Dex2VM to eight different kinds of apps, with more than 1 million downloads from Google play. We are limited to 8 apps because we have to verify the correctness manually. Although the set is small, the types cover quite different kinds of apps in the real-world, including services, games, social, and other major categories. They are shown in Table III.

TABLE III
REAL-WORLD APPS OF DIFFERENT CATEGORIES WITH MORE THAN 1 MILLION DOWNLOADS FROM GOOGLE PLAY.

App	Description	Size(MB)
AndroMoney	keeping accounts	13.5
KakaoBus	smart way to ride a bus	17.9
Wandoujia	Android app store in China	12.1
Khan Academy	free online courses	21.8
pedometer	step counts	2.8
klara weather	weather Forecast	4.7
MX player	Android Media Player	26.8
Swipe Brick Breaker	puzzle game	19.9

Starting with Android 5.0 (Lollipop), ART has completely replaced Dalvik VM as the runtime system for Android [1]. ART runs applications through direct machine code, unlike the existing Dalvik VM-based runtime system, which does so by running the applications Dalvik bytecode through the interpreter [62], [6], [43], [44], [5]. So, in this paper, all Cost related experiments are performed on versions 4.4.4r1 with Dalvik, 5.0 with ART, and 9.0 which is the latest version of Android. Our mobile device is LG Nexus 5 equipped with a 4x Qualcomm* Krait 400 2.3GHz CPU. We install android 4.4.4r1 and android 9.0 on it separately. To measure compatibility, we also create a virtual device for Nexus 5X with version 5.0 and x86 64-bit CPU/ABI on Android Studio 3.5 Canary 13 emulator.

Especially, due to the large version span from 4.4.4r1 to 9.0, the two original apps of Wandoujia and MX player cannot be installed on android 9.0. Therefore, on Android 9.0, we only did experiments with the other six apps.

B. Resilience

1) Manual Attack: In this section, from the attacker's point of view, we use manual work to reverse analysis on Dex2VM protected codes. First of all, we decompile the Android application using a tool named JEB. From Figure 9 and Figure 10, we can see that the logic of the unprotected code is visible and the protected code has only the declaration of methods in the Java layer. At this time, as an experienced attacker will find that the lib directory is added to the protected file directory, and this directory is used by Dex2VM for saving binary files of different ABIs(Application Binary Interface). A binary reverse tool, such as IDA, will be used for further analysis. Combined with the function declaration in the java file, we quickly find implementation of the function by looking at the pseudo-codes outputted from IDA.

```
1  public int onStartCommand(@Nullable Intent arg3,int
       arg4,int arg5){
2    Object v4 = this.mMonitor;
3    _monitor_enter(v4);
4    try {
5      this.mStartIds.add(Integer.valueof(arg5));
6      this.mLastStartId = arg5;
7     _monitor_exit(v4);
8    }
9    catch(Throwable v3){
10     try {
11       lable_14:
12       _monitor_exit(v4);
13         }
14     catch(Throwable v3){
15       goto lable_14;
16     }
17     throw v3;
18   }
19 }
```

Fig. 9. Java code decompiled from the original code using JEB. The attacker can get both the name and body of the function.

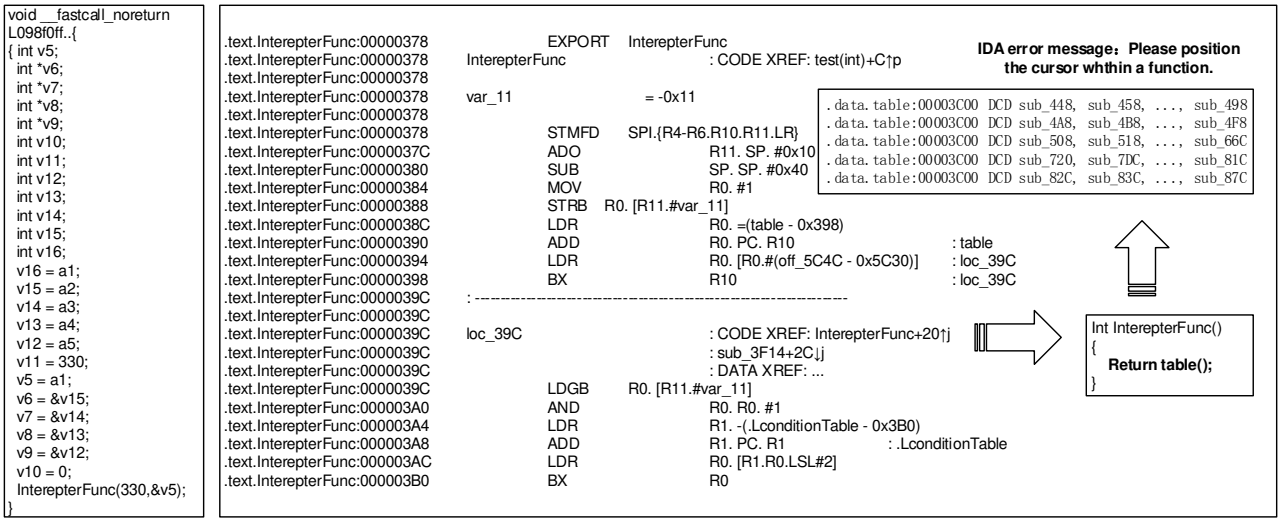
```
1  public native int onStartCommand(@Nullable Intent
       arg1,int arg2,int arg3) {
2  }
```

Fig. 10. Java code decompiled from the virtualized code using JEB, only the registered name of the function is retained.

As can be seen from Figure 11(a), these pseudo-codes are challenging to understand. Secondly, we dynamically track the local execution, trying to find something useful. Figure 11(b) is the function implementation for InterpreterFunc(), which is the interpreter of Dex2VM. However, none of them has complete semantics. Thirdly, we try to piece together the semantic information of the function through dynamic debugging over and over again, but it is invalid. That's because Dex2VM implements multiple sets of virtualization solutions for 27 instructions, and randomly selects one of them to replace the original instructions. And this mapping relationship is amplified and optimized by the LLVM compiler. Ideally, an attacker could get the application raw logic in this way, but it will take enormous time and effort.

2) General Unpacker Tool: In this section, we select six common unpacker tools on android and use them to analyze the Dex2VM protected applications. kisskiss [55] relies on the magic number as a signature to dump odex objects. But it does not work with unknown new packers or even the upgraded version of existing packers. ZjDroid [21] relies on Xposed [36] and locates the DEX files by hooking BaseDexClassLoader to obtain DexOrJar at Jave level, which can be easily detected and interrupted by advanced. Further, since ZjDroid waits for

[1]https://source.android.com/devices/tech/dalvik/



(a) disassembly code of a function (b) disassembly code of InterepterFunc

Fig. 11. Interpreter that can be viewed by using IDA, but IDA can't recognize its contents and reports an error message.

user commands to dump the DEX files, it may be evaded by packers that destroy some critical data used only once. DexHunter [66] recovers DEX files from packed apps in both Dalvik and ART runtime. But it can not handle packers with code obfuscation and junk instructions. Moreover, it only considers the dynamic loading conducted when an app is executed, with a prerequisite that most packers do so to shorten launch time. PackerGrind [61] can successfully reveal the packers protection mechanisms and recover the Dex files with low overhead. While, it cannot decide which code is real if packed apps load different code into the same memory and execute them under different conditions. TIRO [58] is a unified deobfuscation framework for Android apps that can deobfuscate runtime-based obfuscation as well as traditional techniques such as reflection or native method invocation. But it cannot extract execution paths within native code since it is limited to Java. Further, it may not be able to cover all targeted paths in code due to static imprecision and complex path constraints. DROIDUNPACK [18] certainly suffers from limited code coverage as it can only dump the code that executes. And since it is built on top of whole-system emulation, packers that enforce anti-emulation techniques will inevitably break the analysis.

As results are shown in Table IV, DROIDUNPACK is the most powerful tool for unpacker. It can take off almost all types of packers, but do nothing on Dex2VM. So we can get a conclusion: the general unpacker tools are invalid for Dex2VM. This is because Dex2VM directly compiles the bytecode in DEX file into native virtual code, rather than restores code at the running time like the traditional packer method. Therefore, it can be concluded that Dex2VM has good resistance to automated attack tools.

3) VMHunt: VMHunt [59] is a state-of-the-art technique to automatically identify and simplify virtualized code sections from an execution trace. It locates the boundary of partially-virtualized code based on an inherent property of standard virtual machine design: context switches occur between virtu-

TABLE IV
UNPACKING TOOLS AND UNPACKING RESULTS. × INDICATES AN ERROR DURING THE UNPACKING PROCESS, √ INDICATES THAT THE PROGRAM CAN BE EXECUTED NORMALLY AFTER THE UNPACKING RESULT IS REPAIRED.

	Overall Packing	Function Packing	DVM ART	Native Packing	Dex2VM
kisskiss	√	×	×	×	×
zjdroid	√	×	×	×	×
Dexhunter	√	√	√	×	×
PackerGrind	√	√	√	×	×
TRIO	√	√	√	√	×
DROIDUNPACK	√	√	√	√	×

alization application and native OS to ensure isolation. It has been proved that VMHunt correctly extracts the virtualized section from the latest version of well-known virtualization obfuscators without false positives, such as Code Virtualizer [51], VMProtect [47], EXECryptor [53], and Themida [52].

To evaluate whether VMHunt is still valid for Dex2VM, we take the above eight Dex2VM protected experimental samples as the input of VMHunt. However, the experimental results show that VMHunt does not extract a virtualized section from any of the samples. Analysis of the principle of VMHunt, we find that it uses a pattern match method to match instruction sequences that push all registers to stack or pop them back, such as push edi, push esi, push ebx, push ebp, push ecx, push eax. Dex2VM is an intermediate represent-based code virtualization protection technology. The virtualized LLVMIR will be compiled as a new ELF file rather than rewrote into the original ELF file like others. There is no such distinct instruction sequence feature. That is why VMHunt extracts zero virtualized section from a Dex2VM protected program.

C. Stealth

Commercial obfuscation tools such as Themida, VMProtect, Code Virtualizer, generally contain a variety of techniques and are challenging to use as reference objects. In this work, we use the following three tools. Tigress [49], [14] is a source-

TABLE V
FREQUENT N-GRAMS IN THE CORPUS. THE VALUES SHOW THE RATIO OF THE FREQUENCY TO THE TOTAL FREQUENCY[%].

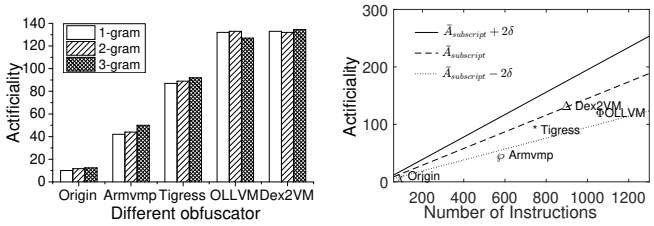
No.	1-gram		2-gram		3-gram	
1	ldr	8.58	ldr-ldr	2.17	ldr-add-add	0.85
2	add	6.87	ldr-add	2.00	str-str-str	0.68
3	mov	6.23	add-add	1.70	add-ldr-add	0.65
4	str	4.18	mov-mov	1.50	ldr-ldr-ldr	0.61
5	bl	4.15	bl-ldr	1.42	ldr-cmp-beq	0.57
6	cmp	3.77	ldr-mov	1.36	add-add-ldr	0.56
7	b	2.06	str-str	1.34	cmp-beq-ldr	0.52
8	beq	1.87	add-bl	1.32	add-bl-ldr	0.46
9	strtmi	1.4	mov-bl	1.28	mov-bl-ldr	0.45
10	stm	1.35	ldr-cmp	1.25	ldr-mov-ldr	0.43

to-source virtualizer, Obfuscator-LLVM [20] is an obfuscator working on respectively LLVM intermediate representation level. To compare the difference between binary-based virtual machines and intermediate language-based virtual machines, we also implemented a virtual machine named Armvmp with the ideas in this article [34] on the arm instruction set.

1) Artificial: N-gram models are usually used in natural language processing to calculate the occurrence probability of a word sequence. Citation [29] proposes a way of using N-gram to calculate the "artificial" of protected code. "Artificial" can effectively evaluate the degree to which protected code can be distinguished from unprotected code. However, it only provides a corpus for some x86 instruction sets. The test mobile phone used in our experiments is based on the arm instruction set. So, we extracted about 10 million instructions from the Android system library as training samples for building a corpus of the arm instruction set. Then we use N-gram models to evaluate the strength of Dex2VM. Table V shows the top10 frequency instruction combinations and their corresponding probability when N is 1, 2, 3 in our corpus.

We use Dex2VM, Tigress, OLLVM, Armvmp to protect the same Android application separately for generating the test objects. According to the corpus provided in Table V, we calculate the artificiality of the protected target codes, respectively, when N is from 1 to 3. The results are shown in Figure 12(a), the artificiality of the obfuscated code of OLLVM and Dex2VM are high. To some extent, artificiality depends on the number of instructions. So, to ensure the accuracy of the experimental results, we calculate the standard artificial situation of each evaluation object. Figure 12(b) shows the artificiality of each evaluation target in the case N=3. The vertical axis represents the artificiality, and the horizontal axis represents the number of opcodes. The Figure 12(b) shows that most of the target codes are within the standard deviation. Dex2VM is the closest to the average.

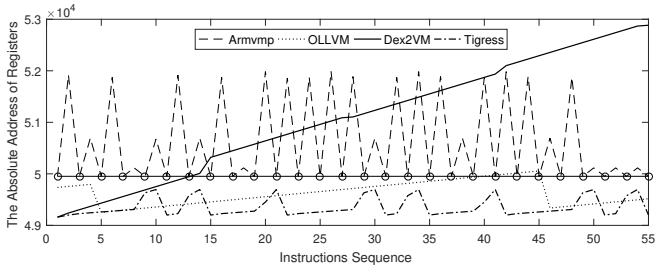
2) Instruction Access Space Sampling: Attackers are very sensitive to the features extracted from the program with high-level information. Runtime address access space is one of the most apparent features. So, in this experiment, we try to find out how much of these features can be exposed to the attacker. We also use four tools to protect the same Android application separately for generating test objects. Figure 13(a) is access space sampling. The vertical axis represents the absolute address of the register, and the horizontal axis represents the



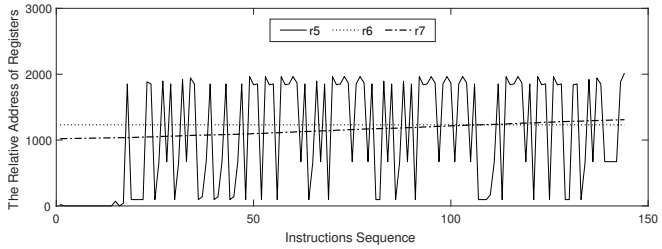
(a) Results for the artificiality of the target code.

(b) The relation between the artificiality and the number of opcodes (N = 3).

Fig. 12. Artificial Results. Tigress, OLLVM and Dex2VM is within the standard deviation of Artificiality. Dex2VM is the closest to the average.



(a) Instruction access space sampling.



(b) Result of sampling the partial registers.

Fig. 13. Access space sampling. Binary-based code virtualizer Armvmp has a significiant cyclic shock. Its register r5 always access a limited address space. The attacker can use this to derive the context of VM.

sequence of instructions. We can see that Armvmp has a significant cyclic shock, which is caused by the repeated call of the dispatcher. The attacker can use this phenomenon to locate the position of the scheduler for implementing the next attack. OLLVM is an obfuscator without dispatcher. The dispatcher of Dex2VM is hidden by the technique used in IV-B3.

To further check the usage of the registers, we use the IDApython script to dye the registers at the virtual machine runtime. Figure 13(b) shows instruction access space sampling for Armvmp. The vertical axis represents the relative address, which is the value of actual access address of the register minus a base number. The horizontal axis represents the sequence of instructions. The register r5 always accesses in a limited address space, the value of register r6 remains unchanged, and register r7 approaches linear growth. Therefore, the attacker can use this information to derive the address table, context of a virtual machine which is a set of spaces used to emulate the actual registers, and the associated register pointers of the virtual instructions.

From the experiments, we can see that Armvmp has distinct loop characteristics. That's because code virtualization for binary code has some limitations. The practical information

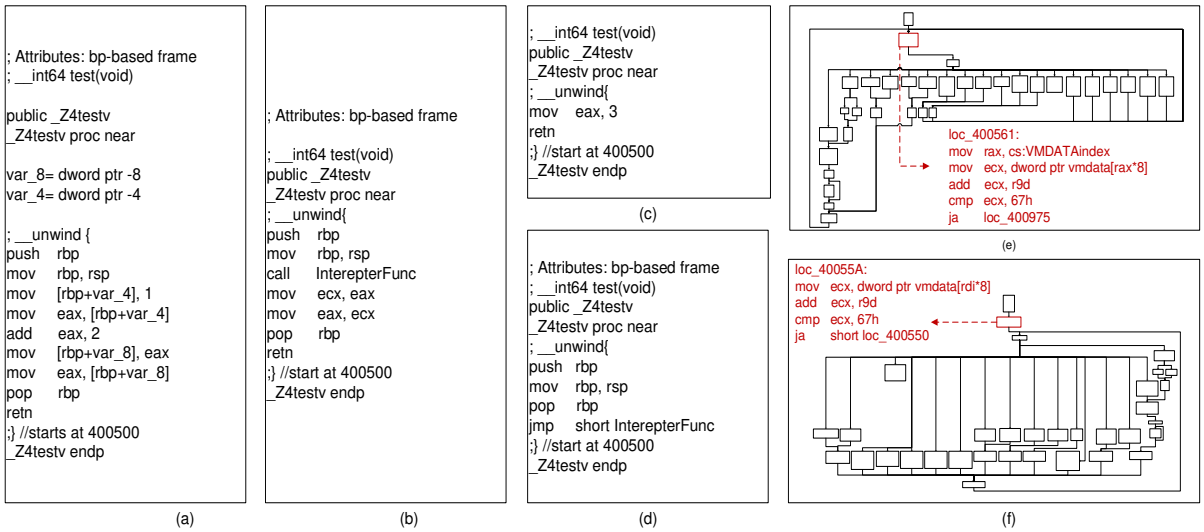


Fig. 14. Experiment result to verify whether optimization affects virtualization in LLVM. All subfigures are produced by IDA. a:original program. b:virtualized code without optimization. c:optimized code with option -o3. d:virtualized code with optimization option -o3. e:InterepterFunc of b. f:InterepterFunc of d. In e and f, zoom in to the red box, it is the disassembly instructions.

such as virtual instructions, dispatcher, handler, an interpreter has to write back into the original binary file. This limitation leads to less work on the hiding of critical information in virtualization, so the interpreter is easily removed by de-virtualization. Our intermediate language-based virtualization can effectively reduce the exposure of sensitive features of protected code. The principle behind it is that the intermediate code is compiled into the target code will lead to a sharp increase in instructions.

3) Optimizations in LLVM: LLVM is a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software [35]. Such lifelong code optimization techniques encompass optimizations at compile-time, link-time, install-time, and runtime. The design of the compile- and link-time optimizers in LLVM permit the use of a well-known technique for speeding up interprocedural analysis. They operate on the LLVM representation directly, taking advantage of the semantic information it contains. LLVM currently includes several interprocedural analyses, such as call graph construction, Mod/Ref analysis, interprocedural transformations like inlining, dead global elimination, dead argument elimination, and so on. So whether our instruction set and the virtual machine will be optimized by the LLVM, this is what we need to demonstrate.

To verify whether our virtualization code is optimized by LLVM at compile-time, we do a set of experiments. First, we write a function called test with Language C. The test function contains three statements: a=1; c=a+2; return c. We respectively compile test by clang/LLVM with o0, -o2, -o3. Second, we implement code virtualization on test with Dex2VM, then choose the Optimization option with o0, -o2, -o3. As we can see from the Figure 14(c), there is only one statement (mov eax, 3) left after LLVM optimized the original program. (b) is the result of code virtualization on the original program without Optimizations. (e) is the call graph of the InterepterFunc function in (b). (d) is the result of code virtualization on the original program with Optimization option o3. The LLVM optimizer converts the function call into a jump table call. Figure 14(f) is the call graph of the InterepterFunc function in (d). From Figure 14 e) and f), we can see that the LLVM optimizer changes the structure of the virtualized code. Then we zoom in to the red box and find that these two boxes have the same instructions pieces, same as other basic blocks. The reason is that the optimizer can not recognize our custom instruction set. If an attacker wants to use LLVM for instruction-level optimization to achieve the purpose of streamlining instructions, he needs to rewrite the specialized backend optimizer.

D. Cost Overhead

1) Benchmark on Dex2C: As mentioned above, Dex2VM contains two parts, Dex2C and C2VM. Dex2C extracts the functions that have less interaction with the system through the Decision-making Model and implements them in the native layer. This process guarantees the security of Dex bytecode and also greatly reduces the overhead of the system. To verify this conclusion, we pre-compile the 0xbench suite with our Dex2C. 0xBench [1] is Google's official test program, 0xlab integrates 17 benchmarks for 0xBenchmark, including computing performance, JavaScript benchmark, 2D graphics rendering, 3D graphics rendering, garbage collection performance test. In our experiment, we use 2D, 3D, and SciMark2. As the experimental results are shown in Table VI, the performance of 2D and 3D have almost no change before and after protection. Because most of their samples use source code from Android SDK sample programs and iPhone SDK sample programs or Android API. So, there is no code that can be translated from the DEX layer to the native layer. But SciMark2 is a composite Java benchmark measuring the performance of numerical codes occurring in scientific and engineering applications. According to our Decision-making Model, some java layer code with high-performance overhead

TABLE VI
RESULTS OF 0XBENCH.

	Case	Android4.4.4r1		Android5.0		Android9.0	
		before	after	before	after	before	after
2D (fps)	Canvas	58.77	58.91	58.16	58.43	58.49	59.36
	Circle	58.17	64.21	55.08	56.13	46.19	67.18
	Circle2	36.53	41.68	33.37	38.52	37.94	41.82
	Rect	12.68	14.13	10.25	12.13	13.44	16.52
	Arc	25.92	30.84	20.60	27.01	26.27	27.99
	Image	48.80	54.46	33.02	39.32	23.09	26.85
	Text	58.41	59.24	58.77	58.53	59.67	59.33
3D (fps)	Cube	55.79	55.70	58.87	54.88	59.28	59.19
	Blending	63.96	64.11	63.41	63.47	63.7	63.68
	Fog	64.06	63.82	63.32	63.60	63.58	63.63
	Teapot	60.60	60.60	60.60	60.60	53.41	55.55
SciMark2 (Mflops)	Composite	143.35	172.82	161.1	186.65	191.09	282.69
	Fast	95.42	102.36	172.08	187.23	206.18	301.56
	Jacobi	347.20	423.70	400.61	466.58	324.51	474.15
	MOnte	11.95	11.30	11.87	12.52	10.57	13.99
	Sparse	110.70	127.21	58.91	79.80	110.26	179.45
	dense	151.47	199.55	162.00	187.13	303.95	444.29

is selected to translate into native-layer code with low overhead. This experimental result proves that the transformation of DEX bytecode to C/C++ code in the pre-compilation stage can indeed improve the execution efficiency of the code.

2) Cost Overhead on Dex2VM: To achieve the measurement of space-time overhead for Dex2VM, we mainly consider the three aspects including CPU occupancy rate, size, and runtime memory usage. In this experiment, each of the selected apps has to run 30 times, and the average value is obtained to represent the experimental results. During the test, the user's behavior is simulated by Monkey [24], which triggers random clicks, slides, text or character input. The time limit for each data collection is set to ten minutes. The interval of random event setting for Monkey is 1000ms, that is, there are at least 600 random inputs in each data collection process. Finally, we use Tencent's open-source performance testing tool GT [54] to obtain the corresponding experimental data. All the experiment data are showed in Table VII.

CPU usage: It can be seen that the CPU usage of the protected program is almost the same as the original app.

Size: As can be seen from the table, compared with the original apps, all the apps protected by Dex2VM contain a modified DEX file with a larger size and a newly generated SO file. Although some functions in the DEX file are implemented in the native layer, the reason why it is still larger than the original one is that the instructions of the original functions are filled with nop, and JNI registration information is added. From the perspective of software complexity, the increased size contains more instructions, which to some extent, increases the difficulty of an attacker's reverse analysis.

Memory usage: After the protection, the total memory usage of the program shows an upward trend, but the increase is not large. More specifically, the memory consumed by the Dalvik virtual machine of the program drops, while the memory space consumed by native code rises, which is consistent with the protection features of Dex2VM.

In summary, it can be seen from the above experimental results that there is a big difference in the performance of the program before and after Dex2VM protection. The main reason is that the native layer directly executes CPU instructions, which is more efficient and straightforward. Therefore, we draw the following conclusions: Firstly, the conversion of D2C reliably improves the efficiency of the program running, effectively balancing the performance overhead introduced by virtualization. Therefore, our scheme is an effective solution for both security and performance issues. Also, Dex2VM has good performance from the data of space-time overhead.

3) Power Consumption: Trepn^{TM} Profiler is an on-target power and performance profiling application for mobile devices, which is developed by Qualcomm. Its principle is that SnapDragon800 + series chips are built into multiple sensors on components, such as CPU core, digital core, power monitoring, etc., to obtain current data directly from the hardware. But some Trepn features stopped working when Android 7.0 was released including direct power reporting. So we use Trepn^{TM} Profiler to collect power consumption of original samples and protected samples on Android 4.4.4r1 and 5.0. As to 9.0, we use $adb\ shell\ dumpsys\ batterystats$ to get the power consumption. The Android framework layer implements the function of power statistics through a system service called batterystats. And adb obtains energy consumption by accessing this system service.

To ensure the correctness of the experimental data, before the experiments, we turn off all the other apps, and the screen brightness is minimized. Also, the test time lasts 10 minutes, and the remaining power of the mobile phone must be higher than 80% during this process. The experiment is repeated three times to average.

As shown in Table VII, when running on Android 4.4.4r1, the power consumption is reduced by 28.5% after protection. When running on Android 5.0, power consumption is reduced by 6%. To explain why there is a big difference in power consumption between the two versions, we do a more in-depth investigation. Starting with Android 5.0 (Lollipop), ART runtime has completely replaced Dalvik Virtual Machine. During the installation of the application, Ahead-of-Time Compiler in ART translates the DEX bytecode into machine code and stores it on the device's memory. This process only happens when the application is installed on the device. JIT compilation is no longer needed, the code executes much faster. So, that's why there is a 22.5% gap between them.

When coming to Android 9.0, we find that the power consumption floating range is between plus and minus 5% with and without Dex2VM. The reason is that the data obtained by the adb is not stable. Because there are two ways for $batterystats$ to get information about battery usage. One is push action that some hardware modules (wifi, Bluetooth) notify $batterystats$ to record the time when the status changes. The other is pull action that $batterystats$ actively records the starting point of some hardware modules(CPU) to calculate the time the activity uses the CPU. So the statistical results of $batterystats$ are affected by the frequency of data collection. However, from the experimental results, we can conclude that the effect of Dex2VM on power consumption is within the user's tolerance.

In general, the introduction of virtual machine protection will cause performance degradation. But our power consumption is lower. First of all, in the pre-compilation phase, we use

TABLE VII
EXPERIMENTAL EVALUATION DATASET AND RESULTS.

Version	Application	CPU(%)		Size(KB)		Memory(KB)									Power(mW)		
						Total			Dalvik			native					
		before	after	before(DEX)	after(DEX+SO)	before	after	increase	before	after	increase	before	after	increase	before	after	increase
Android4.4.4r1	AndroMoney	42.15	43.81	5019	5442+2487	69298	73845	6.56%	11186	12905	15.37%	4461	5071	13.67%	904.53	615.87	-31.91%
	KakaoBus	32.04	33.78	6820	7380+2226	80462	83951	4.34%	23793	22951	-3.54%	3240	4254	31.30%	876.58	606.93	-30.76%
	Wandoujia	34.43	35.44	4449	4761+2719	39874	42087	5.55%	5439	3217	-40.85%	4605	6700	45.49%	981.64	651.82	-33.60%
	Khan Academy	32.27	36.33	9350	10007+737	110971	125976	13.52%	12702	13989	10.13%	11851	14438	21.83%	942.43	682.26	-27.61%
	pedometer	34.19	20.78	3205	3579+681	83208	85309	2.52%	11282	9800	-13.14%	4550	6000	31.87%	888.56	621.09	-30.10%
	klara weather	36.72	34.22	6228	6848+719	68231	71868	5.33%	10060	9809	-2.50%	2973	3970	33.54%	822.44	657.54	-20.05%
	MX player	44.97	47.66	18391	19734+2591	41579	51969	24.99%	12151	13503	11.13%	2963	10100	240.87%	914.74	653.53	-28.56%
	Swipe Brick Breaker	38.23	41.19	5571	6015+144	94578	95370	0.84%	2313	2513	8.65%	2313	31899	1279.12%	2097.93	1573.18	-25.01%
Android5.0	AndroMoney	27.7	36.32	5019	5442+2487	143873	191249	32.93%	33678	30337	-9.92%	200931	25111	-87.50%	644.64	634.94	-1.50%
	KakaoBus	20.83	29.34	6820	7380+2226	94395	129800	37.51%	20865	20734	-0.63%	13085	20235	54.64%	683.98	624.65	-8.67%
	Wandoujia	42.89	44.61	4449	4761+2719	151928	15686	-89.68%	28365	23380	-17.57%	27370	30211	10.38%	704.55	658.72	-6.50%
	Khan Academy	32.28	38.39	9350	10007+737	263860	336925	27.69%	48057	33318	-30.67%	55089	71985	30.67%	758.63	700.49	-7.66%
	pedometer	20.06	20.78	3205	3579+681	88527	112451	27.02%	25808	23414	-9.28%	12771	14593	14.27%	663.15	628.54	-5.22%
	klara weather	20.62	21.94	6228	6848+719	144591	155514	7.55%	35737	34527	-3.39%	18781	20750	10.48%	737.45	697.34	-5.44%
	MX player	19.07	20.52	18391	19734+2591	161569	176764	9.40%	33980	32919	-3.12%	28650	36577	27.67%	686.85	651.11	-5.20%
	Swipe Brick Breaker	45.48	46.16	5571	6015+144	112024	113534	1.35%	5896	6133	4.02%	40737	49544	21.62%	1733.52	1607.7	-7.26%
Android9.0	AndroMoney	15.56	17.56	5019	5442+2487	69765	71150	1.99%	2269	2131	-6.08%	28455	31307	10.02%	926.63	977.36	5.47%
	KakaoBus	15.15	11.2	6820	7380+2226	53895	79252	47.05%	5014	3089	-38.39%	15917	40754	156.04%	579.56	595.53	2.76%
	Wandoujia	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Khan Academy	16.4	26.5	9350	10007+737	141704	146055	3.07%	9958	8182	-17.83%	51579	58065	12.57%	693.08	665.77	-3.94%
	pedometer	43	4.88	3205	3579+681	48681	47024	-3.40%	1582	1499	-5.25%	15112	15654	-3.59%	592.27	581.49	-1.82%
	klara weather	15.99	10.34	6228	6848+719	71777	75522	5.22%	2639	2304	-12.69%	27960	25830	-7.62%	577.35	584.46	1.23%
	MX player	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Swipe Brick Breaker	25.04	29.73	5571	6015+144	70449	70700	0.36%	1368	1364	-0.29%	35561	39173	10.16%	1212.59	1200.89	-0.97%

TABLE VIII
THE FUNCTIONALITY RATING.

App	Install time	Startup time	Function usability	Action response	Flash back
AndroMoney	3.5	4.5	5	5	5
KakaoBus	3.5	4.5	5	4	5
Wandoujia	2.5	4.5	4.5	4.5	5
Khan Academy	3.5	4	3.5	4	3
pedometer	3.5	4.5	5	5	4
klara weather	3.5	4.5	4	5	5
MX player	4	4	2.5	4	1.5
Swipe Brick Breaker	4	4.5	5	4	5
Avg	3.50	4.38	4.31	4.44	4.19

the Decision-making Model to filter those functions in DEX bytecode that interacting less with the context, then replace them into c code in the native layer, which reduces the performance overhead to some extent. Secondly, in compile-time, we only virtualized 20% of the codes rather than all codes. Therefore, from this experiment, we can get a conclusion that the power consumption is reduced by 6% after protected by Dex2VM.

E. Functionality Study

The evaluation on the correctness is limited by that there is no way to check function or semantic equivalence between obfuscated code and original code, unless manually checking. In this experiment, we perform user study to quantify the impact of security features on user experience, including installation time, startup time, availability of features, the response speed of actions, and flashback.

Our user study has 16 participants. All of them are at the age group of under the 30s and are familiar with the Android system. We give each participant a half-hour experience time to use a group of original and protected apps without marking. Experience content includes app installation and function use. In the user study, we ask each participant to rate the functionality of apps on a 5-point Likert-scale, where 1 = very poor and 5 = excellent.

Table VIII is a functionality rating result in our user study. Each app has two user ratings, and we only record the average. The apps install time has the lowest user ratings, which is 3.50. This is because when the application is first installed, the DEX bytecode is precompiled into native code. Although this process takes some time, it only happens once and does not affect the startup and execution time of the application. In the flashback item, two users scored 2 points and 1 point for the MX player. The reason for this is that a pointer in the protected app does not release the memory in time. Now we have fixed this bug. Overall, this user study shows that Dex2VM can guarantee the availability of features while balancing security and performance.

It is worth motioning that although code obfuscation has been actively researched for quite a long time, how to systematically measure the effectiveness of an obfuscator remains an open problem. It has been proven that a perfect obfuscator does not exist [10]. Thus, there is a widely accepted consensus that the goal of obfuscation is to protect the code by making reverse engineering so technically difficult that it becomes impossible or at the very least economically inviable [12]. In this paper, the experiment result shows that Dex2VM is a novel tool which possesses strong security strength and good stealth, with only modest cost.

VI. DISCUSSIONS

In this section, we discuss Dex2VM's limitations, potential countermeasures, and future work. First, the virtualized function will eventually interact with the Android runtime anyway. The attacker may use a JNI hooker to listen to the call, and combine the manual analysis method to guess the purpose of the program. Although the manual analysis process is very time-consuming, this interactive feature gives attacker the possibility to restore the program bytecode. We can defend this attack to design a lightweight Hook detection mechanism.

Theoretically, our performance experiment results are unlikely to be so good if all the C codes extracted from the DEX file are virtualized, instead of only taking 20% of them to be virtualized. However, we wish to reiterate that whole

program virtualization rarely happens in practice. Complete program virtualization translates the entire program to VM instructions and interprets them during runtime, which will cause a significant slowdown. In future work, we will discuss performance changes when the scale of the virtualization codes reaches 40%,60%, and 80%.

In future work, we can also optimize the disassembly engine of the Dex2VM to translate directly from dex code to LLVMIR. In this way, the error rate in the disassembly process can reduce, and the reduction of the conversion link minimizes the possibility of reverse engineering by the attacker. Also, the packing technique is a double-edged sword for both legitimate and malicious apps. From a design point of view, we should think about how to avoid it being abused by attackers.

VII. Related Work

Obfuscator-LLVM(OLLVM): It is an open-source code obfuscator based on LLVM framework [37], [20], [26]. The whole project contains three relatively independent LLVM passes. Each pass implements a kind of obfuscation, and they are Instructions Substitution, Bogus Control Flow, Control Flow Flattening. Through these obfuscation methods, the original program flow or part of them can be blurred, bringing some difficulties to the reverse analysis. Since the above three passes are implemented based on LLVMIR, in theory, this obfuscator supports any language and machine architecture in the world.

OLLVM only changes the grammatical structure and control flow structure of the program. Therefore, there is a possibility of deobfuscation by symbol execution [41], [32]. Dex2VM is a fine-grained code protection method, which acts at the instruction level. So even if the attacker gains the program structure by symbol execution, the custom-defined instructions cannot be recognized.

Packers and Unpackers: At present, function-level obfuscation and encryption [30] at both DEX and native levels are the mainstays of the android packer in industry, such as Bangcle [8], ijiami [39], Qihoo [2], Baidu [7], Tecent [50], and so on. However, recent studies have shown that it is feasible to recognize different types of these commercial packers by signatures and recover the DEX file by modifying DVM to hook certain important functions or dump Dalvik data structures [66], [64], [18], [48], [61], [58]. So far, there has been a small amount of work trying to use code virtualization in Android packers. Divilar [68] transforms its Dalvik bytecode into a randomly generated intermediate language and wraps the resulting binary together with a lightweight virtual instruction interpreter. Xu [60] transplantes the traditional UPX packing technology to the ARM architecture for native code protection. Collberg proposes the Tigress obfuscator [49], [14], which is a source-to-source-level obfuscation transformation tool based on Ocaml [40] and only supports the C99 standard language. However, code virtualization at the Java layer has a limitation that the newly generated DEX file with the open file format and semantic meaning makes the decode-dispatch pattern more exposed. Dex2VM overcome both of these three limitations with pre-compilation and compile-time code virtualization based on the LLVM framework.

Although packers have been well studied, a series of solutions have been proposed to defeat them [18], [55], [21], [66], [64], [61], [58]. According to design choices of extracting code, current Android unpackers can be categorized into four types: 1) signature-based memory dump unpacker; 2) hooking-based memory dump unpacker; 3) DEX file assembly unpacker; 4) whole-system emulation based unpacker. Most of them focus on various commercial packers, barely recognizing and analyzing the unknown customer packing technique. Moreover, they all take an assumption that there is an apparent boundary between packer's code and the original code. However, Dex2VM is a virtualization of the intermediate language, it is finally compiled into native code with the help of the LLVM compilation framework. So it can overcome this limitation since there are no clear boundaries. Finally, few of them is able to recognize the unknown packing technique, and understand what happens at the native level, let alone the interactions between Java and native. DROIDUNPACK [18] is a state-of-the-art unpacker that has a whole view in multiple levels of the system and can detect unknown packers. Although it can dump codes that execute in Dex2VM, it can not recover the semantic information from custom-defined virtualization instructions.

Deobfuscation of code virtualization: Automatic deobfuscation tools were proposed that could recover the original functionality [11], [28]. The traditional approaches are either static or dynamic analysis [27], [31]. Recent approaches for deobfuscation use techniques based on taint analysis and symbolic execution [16], [45], [63], [59]. Both of them have drawbacks. Taint tracking-based solution requires an IR transformation of the binary instructions and produce a computational overhead by design, due to the tracking and simulation of memory operations [16], [17], [63]. Symbolic execution is the handling of path explosions. Depending on the symbolic variables, several additional paths need to compute subsequently. These results in an increased number of computations [63].

There is a widely accepted consensus that developing an obfuscation scheme resilient to all reverse engineering threats is too ambitious [9]. If the attacker invests enough time and energy, he will eventually be able to crack the virtualized code. Hence, making reverse engineering more difficult (but not impossible) could be a more realistic goal to pursue. All the experimental results show that Dex2VM possesses strong security strength and good stealth, with only modest cost.

VIII. Conclusion

When preventing reverse engineering from infringing intellectual property on smartphone devices, vulnerability and performance are two big challenges due to a decode-dispatch pattern and energy limitations. In this paper, we present a double-layer packer with pre-compilation as the first layer and compile-time code virtualization as the second layer. By utilizing the certain design and features of the LLVM framework, we can overcome the limitation of vulnerability and performance for code virtualization on the Android platform. We implement Dex2VM, a tool that translates the DEX bytecode

into the common LLVM intermediate representations where a unified code virtualization pass can be applied. We evaluate Dex2VM concerning resilience, stealth, cost, and functionality on eight representative Android applications. The experimental results show that the proposed approach can effectively protect the target code against a state-of-the-art unpacking tool and code reverse engineering tool that is specifically designed for code virtualization, and it achieves at a good stealth and the cost of little overhead of memory, code size, and energy consumption.

REFERENCES

[1] 0XLAB. 0xbench integrated android benchmark suite by 0xlab, 2018. https://code.google.com/archive/p/0xbench/.

[2] 360. Qihoo, 2019. http://jiagu.360.cn.

[3] AIKAWA, M., TAKARAGI, K., FURUYA, S., AND SASAMOTO, M. A lightweight encryption method suitable for copyright protection. vol. 44, IEEE Press, pp. 902–910.

[4] ANTHONY SCARSELLA, W. S. Worldwide smartphone forecast update, 20192023, 2019. https://www.idc.com/getdoc.jsp?containerId=US45235019.

[5] ARNATOVICH, Y. L., WANG, L., NGO, N. M., AND SOH, C. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. vol. 6, pp. 12382–12394.

[6] BACKES, M., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Poster: Towards compiler-assisted taint tracking on the android runtime (art). In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1629–1631.

[7] BAIDU, 2019. https://app.baidu.com/.

[8] BANGCLE. Bangcle, 2017. https://dev.bangcle.com/.

[9] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference* (2001), Springer, pp. 1–18.

[10] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im)possibility of obfuscating programs. In *Advances in Cryptology — CRYPTO 2001* (Berlin, Heidelberg, 2001), J. Kilian, Ed., Springer Berlin Heidelberg, pp. 1–18.

[11] CAZALAS, J., MCDONALD, J. T., ANDEL, T. R., AND STAKHANOVA, N. Probing the limits of virtualized software protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop* (New York, NY, USA, 2014), PPREW-4, ACM, pp. 5:1–5:11.

[12] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial* (01 1997).

[13] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, ACM, pp. 184–196.

[14] COLLBERG C. S., MARTIN S., M. J. E. A. The tigress diversifying c virtualizer, 2018. http://tigress.cs.arizona.edu.

[15] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 275–284.

[16] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 275–284.

[17] COOGAN, K. P. Deobfuscation of packed and virtualization-obfuscation protected binaries. The University of Arizona.

[18] DUAN, Y., ZHANG, M., VASISHT BHASKAR, A., YIN, H., PAN, X., LI, T., WANG, X., AND WANG, X. Things you may not know about android (un)packers: A systematic study based on whole-system emulation.

[19] ERTL, M. A., AND GREGG, D. The behavior of efficient virtual machine interpreters on modern architectures. In *European Conference on Parallel Processing* (2001), Springer, pp. 403–413.

[20] GITHUB. obfuscator-llvm, 2017. https://github.com/obfuscator-llvm/obfuscator/wiki.

[21] GITHUB. Zjdroid, 2019. https://github.com/halfkiss/ZjDroid.

[22] GOOGLE. Google cpu profiler tool, 2017. https://developer.android.com/studio/profile/cpu-profiler.

[23] GOOGLE. Android ndk, 2018. https://developer.android.google.cn/ndk/index.html.

[24] GOOGLE. Monkey, 2018. http://developer.android.com/intl/zh-TW/guide/developing/tools/monkey.html.

[25] JESUSFREKE. baksmali: an disassembler for the dex format, 2019. https://github.com/JesusFreke/smali.

[26] JUNOD, P., RINALDINI, J., WEHRLI, J., AND MICHIELIN, J. Obfuscator-llvm: Software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection* (Piscataway, NJ, USA, 2015), SPRO '15, IEEE Press, pp. 3–9.

[27] KALYSCH, A., GÖTZFRIED, J., AND MÜLLER, T. Vmattack: Deobfuscating virtualization-based packed binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (New York, NY, USA, 2017), ARES '17, ACM, pp. 2:1–2:10.

[28] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode* (New York, NY, USA, 2007), WORM '07, ACM, pp. 46–53.

[29] KANZAKI, Y., MONDEN, A., AND COLLBERG, C. Code artificiality: A metric for the code stealth based on an n-gram model. In *Proceedings of the 1st International Workshop on Software Protection* (Piscataway, NJ, USA, 2015), SPRO '15, IEEE Press, pp. 31–37.

[30] KIM N. Y., SHIM J., C. S. E. A. Android application protection against static reverse engineering based on multidexing. *J. Internet Serv. Inf. Secur. 6(4)* (2016), 54–64.

[31] KINDER, J. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering* (Washington, DC, USA, 2012), WCRE '12, IEEE Computer Society, pp. 61–70.

[32] KING, J. C. Symbolic execution and program testing. vol. 19, ACM, pp. 385–394.

[33] KUANG, K., TANG, Z., GONG, X., FANG, D., CHEN, X., AND WANG, Z. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Computers & Security 74* (2018), 202–220.

[34] KUANG, K., TANG, Z., GONG, X., FANG, D., CHEN, X., XING, T., YE, G., ZHANG, J., AND WANG, Z. Exploiting dynamic scheduling for vm-based code obfuscation. IEEE Trustcom/BigDataSE/ISPA, pp. 489–496.

[35] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[36] LEASWEB. xposed, 2019. https://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053.

[37] LIM, K., JEONG, J., CHO, S.-J., CHOI, J., PARK, M., HAN, S., AND JHANG, S. An anti-reverse engineering technique using native code and obfuscator-llvm for android applications. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems* (New York, NY, USA, 2017), RACS '17, ACM, pp. 217–221.

[38] LLVM-ADMIN TEAM, 2019. https://llvm.org/.

[39] LTD, B. Z. Y. W. A. T. C. ijiami, 2018. http://www.ijiami.cn/.

[40] OCAML. Ocaml is an industrial strength programming language, 2018. http://ocaml.org.

[41] QUARKSLAB. Deobfuscation: recovering an ollvm-protected program, 2014. https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

[42] RAYS, H. Ida pro, 2019. https://www.hex-rays.com/.

[43] SABANAL, P. State of the art exploring the new android kitkat runtime. In *Hack In The Box Security Conference* (2014).

[44] SABANAL, P. Hiding behind art. In *BlackHat Asia* (2015).

[45] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. pp. 94 – 109.

[46] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[47] SOFTWARE, V. Vmprotect software protection. http://vmpsoft.com, year = 2017,.

[48] SUN, C., HUAN, Z., QIN, S., HE, N., QIN, J., AND PAN, H. Dexx: A double layer unpacking framework for android. *IEEE Access PP* (10 2018), 1–1.

[49] TAYLOR, C., AND COLBERG, C. A tool for teaching reverse engineering. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)* (Austin, TX, 2016), USENIX Association.

[50] TECENT, 2019. http://legu.qcloud.com/.

[51] TECHNOLOGIES, O. Code virtualizer: Total obfuscation against reverse engineering, 2017. http://oreans.com/codevirtualizer.php.

[52] TECHNOLOGIES, O. Themida: Advanced windows software protection system, 2017. https://www.oreans.com/themida.php.

[53] TECHNOLOGY, S. Execryptor: Bulletproof software protection. http://www.strongbit.com/execryptor.asp, year = 2017,.

[54] TENCENT. Creat tit, 2018. https://github.com/TencentOpen/GT.

[55] TIM DIFF STRAZZERE, J. J. C. S. Android hacker protection level 0, 2014. https://www.defcon.org/images/defcon-22/dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level-UPDATED.pdf.

[56] WANG, C., HILL, J., KNIGHT, J. C., AND DAVIDSON, J. W. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)* (Washington, DC, USA, 2001), DSN '01, IEEE Computer Society, pp. 193–202.

[57] WANG, C.-S., PEREZ, G., CHUNG, Y.-C., HSU, W.-C., SHIH, W.-K., AND HSU, H.-R. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems* (2011), ACM, pp. 15–24.

[58] WONG, M. Y., AND LIE, D. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, Aug. 2018), USENIX Association, pp. 1247–1262.

[59] XU, D., MING, J., FU, Y., AND WU, D. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, ACM, pp. 442–458.

[60] XU, J., ZHANG, L., SUN, Y., LIN, D., AND MAO, Y. Toward a secure android software protection system. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing* (Oct 2015), pp. 2068–2074.

[61] XUE, L., LUO, X., YU, L., WANG, S., AND WU, D. Adaptive unpacking of android apps. In *Proceedings of the 39th International Conference on Software Engineering* (Piscataway, NJ, USA, 2017), ICSE '17, IEEE Press, pp. 358–369.

[62] YADAV, R., AND BHADORIA, R. S. Performance analysis for android runtime environment. In *2015 Fifth International Conference on Communication Systems and Network Technologies* (April 2015), pp. 1076–1079.

[63] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 674–691.

[64] YANG, W., ZHANG, Y., LI, J., SHU, J., LI, B., HU, W., AND GU, D. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses* (Cham, 2015), H. Bos, F. Monrose, and G. Blanc, Eds., Springer International Publishing, pp. 359–381.

[65] YUSCHUK, O. Ollydbg, 2014. http://www.ollydbg.de/.

[66] ZHANG, Y., LUO, X., AND YIN, H. Dexhunter: Toward extracting hidden code from packed android applications. In *Computer Security – ESORICS 2015* (Cham, 2015), G. Pernul, P. Y A Ryan, and E. Weippl, Eds., Springer International Publishing, pp. 293–311.

[67] ZHAO, J., NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, ACM, pp. 427–440.

[68] ZHOU, W., WANG, Z., ZHOU, Y., AND JIANG, X. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2014), CODASPY '14, ACM, pp. 199–210.