IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 4, 2019, accepted November 4, 2019, date of publication November 19, 2019, date of current version December 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2954165

Invalidating Analysis Knowledge for Code Virtualization Protection Through Partition Diversity

WEI WANG[1,2], MENG LI[1,2], ZHANYONG TANG[1,2], HUANTING WANG[1,2], GUIXIN YE[1,2], FUWEI WANG[1,2], JIE REN[3], XIAOQING GONG[1,2], DINGYI FANG[1,2], AND ZHENG WANG[4]

[1]School of Information Science and Technology, Northwest University, Xi'an 710127, China
[2]Shaanxi International Joint Research Centre for the Battery-Free Internet of Things, Northwest University, Xi'an 710127, China
[3]School of Computer Science, Shaanxi Normal University, Xi'an 710062, China
[4]School of Computing, University of Leeds, Leeds LS2 9JT, U.K.

Corresponding author: Zhanyong Tang (zytang@nwu.edu.cn)

This work was supported in part by the NSFC under Grant 61972314 and Grant 61672427, in part by the Key Research and Development Project of Shaanxi Province under Grant 2017GY-191, in part by the International Cooperation Project of Shaanxi Province under Grant 2019KW-009 and Grant 2017KW-008, in part by the Shaanxi Science and Technology Innovation Team Support Project under Grant 2018TD-O26, in part by the Foundation of Education Department of Shaanxi Province Natural Science under Grant 15JK1742, and in part by the Ant Financial through the Ant Financial Science Funds for Security Research.

ABSTRACT To protect programs from unauthorized analysis, virtualize the code based on Virtual Machine (VM) technologies is emerging as a feasible method for accomplishing code obfuscation. However, in some State-of-the-art VM-based protection approaches, the set of virtual instructions and bytecode interpreters are fixed across the whole programs. This means an experienced attacker could extract the mapping information between virtual instructions and native code from programs, and use this knowledge to uncover the mapping relationships in similar protecting applications. To address this problem, we present CoDiver (Code Virtualization Protection with Diversity), a novel VM-based code obfuscation system in this paper. The main idea of our approach is to obfuscate the mapping between the opcodes of bytecode instructions and their semantics. To achieve this goal, we first turn every protected code region into multiple parts by partition proceeding, randomize the mapping of opcodes and their semantics of each part. By this way, we could translate the bytecode instruction into different native code in different sections of the obfuscated code. This method could increase the diversity of program behavior significantly. As a result, it will be useless to learn the mapping relationship between bytecode and native code of some other programs, then migrate it into a new program. We build a prototype of CoDiver and tested it on a set of real-world applications. Experimental results show that as compared with two state-of-the-art VM-based code obfuscation approaches, our approach is more effective and could provide stronger protection with comparable runtime overhead and code size.

INDEX TERMS Instruction set randomization, reverse engineering, virtualized obfuscation.

I. INTRODUCTION

For software developers, unauthorized code reverse engineering is an important threat. This could bring up various attacks, taking out advertisements from the application, including removing copyright protection of software or injecting malicious code into the program. Code obfuscation could make it difficult to trace and analyze the program, protect the software from unauthorized code modification [1]–[7].

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Esposito.

Code virtualization based on Virtual Machine (VM) is emerging as a promising way for implementing code obfuscation [8]–[11]. It converts the opcodes into another code that has an identical function to the original code. This strategy forces the attacker to face with an unfamiliar instruction set, which can significantly increase the time and effort involved in the attack. As an attacker implement reverse engineering of VM-obfuscated code, it will typically follow two steps. First, the attacker needs to understand the semantics of individual bytecode instructions by analyzing the virtual interpreter. Then, they would understand the program logic by

translating the bytecode back into native machine instructions or even high-level program languages [12], [13]. In these two steps, the difficulty varies at different stages, and the most-consuming process is the first one that understanding the semantics of individual bytecode instructions. This step also involved in analyzing the handler that will be used to interpret every bytecode instruction.

Researchers have proposed numerous approaches to protect VM handlers from reverse engineering. Most of the studies focus on how to increase the diversity of program behavior by obfuscating the handler implementation [14] or using different interpretation techniques to transform a single program through multiple iterations [15], [16]. However, these previous works use a static strategy to convert each native code into a fixed set of bytecode. Which means there are no significant changes in the probability distribution of the native code, the frequency of a bytecode will be present by its handler, but only the expression has changed. So when we use the same obfuscation technique, the attacker could extract the mapping relationship and distribution law from one program, then reuse this knowledge (termed analysis knowledge) to launch the attack on another program.

In this paper, we focus on addressing the challenge of reusing analysis knowledge. We propose CoDiver, a Code Virtualization Protection with Diversity, which can enhance VM-based code obfuscation. We introduce Instruction Set Randomization (ISR) [17] to change the *opcodes* of bytecode instructions and their semantics randomly. As a result, the mapping relationships between bytecodes and their handlers varies across the program. But as we mentioned, this is not sufficient to provide adequate protection, for the distribution of the bytecode instructions have not changed. So further than that, CoDiver partitions the protected code region into several parts and use different mapping mode for the bytecode instructions and their handlers in each part. By doing this, the same bytecode instruction in different parts of the program will have different semantics, so the analysis knowledge of the other program will be void.

The key contribution our paper makes is proposing a corresponding countermeasure to block the reuse of analytical knowledge in code reverse engineering. We have build up a prototype of CoDiver, and compare it with Themida [10] and VMProtect [9] while working on a set of real-world algorithms and applications. The test results prove that as compared with two commercial code obfuscation tools, CoDiver could provides stronger protection with similar code size and runtime overhead.

II. BACKGROUND

The virtualization technique could provide convenience in software protection, help us to keep the programs safe from unauthorized analyses. There are a number of VM-based protection tools that have emerged, including themida [10], Code Virtualizer [8] and VMProtect [9]. These VM-based tools performing the protection by converting the protected code area into a set of custom virtual instructions that will be stored in program binaries as bytecodes. At run time, the virtual instructions will revert to native code by byte interpreters.

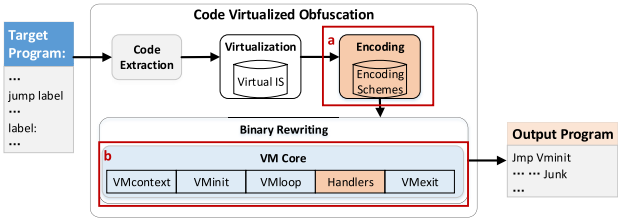


FIGURE 1. The architecture of VM-based code obfuscation. The main work of CoDiver is to strengthen the two steps of VM-based protection (these two steps been marked as "a" and "b"). In the step (a), we partition the protected code into different parts, and obfuscate the `bytecode handlers` to form multiple implementations for each `handler`. In the step (b), we use various obfuscation and anti-taint analysis technologies to protect the important components of the VM core.

As shown in figure 1, it's a classical architecture of a VM-based obfuscation system. Virtual IS (Instruction Set) is the kernel part of this system. The interpreter will been used to convert IS into native code of Virtual instruction interpretation according to the classical *decode-dispatch* method [18], using a set of `handlers` and a `VMloop`. In this system, `VMloop` is the execution engine to fetch and decode the bytecode instructions and dispatch `handler` to interpret the instructions. `VMcontext` involves virtual registers and flags that are hardware-independent. During run time, the virtual registers and flags will be mapped to the underlying hardware, and `VMInit` will initialize the `VMcontext` and save the native context. Accordingly, while exiting VM, `VMExit` will restore the native context. At last, all these VM components will be assembled into a new section and appended to the end of the target program by binary rewriting.

This paper starts with two critical components of the VM-based obfuscation architecture, and highlights them with label "a" and "b", as shown in figure 1. In our method, the area of protected code will be divided into different sections. Then using code obfuscation techniques to generates multiple implementations for each bytecode handler. For the same bytecode handler, there will be different implementations that are semantically equivalent. And these implementations could generate the same output for a given virtual instruction, but during the runtime their execution paths and behaviors are different. To further enhance the protection, we use some obfuscation and anti-taint analysis technologies to protect these key components of the VMCore.

III. THE THREAT MODEL

We use white-box attack [19], [20] as the threat model, which assumes that the attacker has a copy of the target program, and could run it in a malicious host environment [21]. In this threat model, the adversary has full access privileges to the system. And they can use any kind of analysis tools, including static and dynamic types such as `OllyDbg` [22], `IDA` [23] and `Sysinternals Suite` [24]. With these tools, the attacher could trace and analyze the instructions,

monitor process memory and the registers, even modify the instruction bytes and control flows during run time, and so on. As demonstrated in previous work [12], these assumptions are reasonable, that usually available for experienced adversaries.

At present, there are two main attack methods for the protection system based on VM, as been described below. In this paper, we assume an adversary could launch the attack with either one or a combination of these methods. The first one is working on the base of virtual execution analysis which was proposed by Rolles and Rolf [13]. This kind of method requires a deep insight into the code virtualization techniques which is used by the obfuscation system. It extracts key bytecodes and handlers by track the execution of the virtual interpreter dynamically and then restores the program logic through analysis and code simplification. Falliere and Nicolas [12] show that performing such analysis is possible [9]. And this attack method is relevant to the basic principle and structure of code virtualization and has been widely used to analyze obfuscated malware.

The second type of attack will analyze the behavior and semantics of the target program first and then perform the attack base on this analysis. This kind of approach could attack the obfuscation methods whether it's base on code virtualization or not. Yadegari *et al.* [25] propose a behavior analysis method. This approach focuses on analyzing the important behavior of the original code but fails to restore the code. Tang *et al.* [26] propose an attack method that working on semantic analysis. It tracks the input values flow with taint propagation, and simplify the logic of the instructions with semantics-preserving code transformations. This approach has extensive applicability but is limited to a small region of code.

IV. MOTIVATION

In figure 2, shows a scenario of reverse analysis in which an attacker could first extract the *analysis knowledge* from a program, then reuse them to attack some other applications that been protected by the same VM-based code obfuscation scheme. We can see four different target programs in this figure that need to be protected, and marked as A, B, C, and D respectively.

As pictured at right, these four target programs all use a uniform set of virtual instructions and bytecode for the handler to perform the protection. And that means an experienced attacker could first obtain the mapping relationship between virtual instructions and bytecode handlers from one program, and then use this knowledge to accomplish the reverse-engineer of the remaining three programs. As we mentioned before, during attacking the VM-based code obfuscation, uncovering the mapping between virtual instructions and native code is usually the hardest and time-consuming process. So if the attacker could reuse the analysis knowledge, the cost involved in the attack will be reduced significantly.

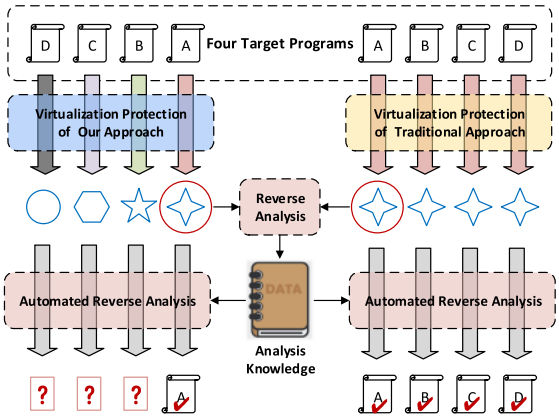


FIGURE 2. The process of code reverse-engineering that reuses the analysis knowledge. Here we marked four different target programs as A, B, C, and D. As pictured at right, every program uses the same code obfuscation scheme which mean a virtual instruction will deterministically mapping to a fixed set of native code. This makes it possible for an attacker to reuse the knowledge obtained from one program to finish the reverse engineer of the other program efficiently. Our approach is shown on the left side, and we can see for different programs the mappings between virtual instructions and native code are totally different. As a result, the attacker no longer able to reuse the previous analysis knowledge to perform the reverse engineering of another program.

Our approach is shown on the left side, the mapping between virtual instructions and native code are varied among different programs. This means the analysis knowledge obtained from one program is inapplicable to the others. This could force the attacker to restart from scratch while performing reverse engineering to a new program. As this example demonstrates, with shuffling the mapping relationship between the virtual instructions and bytecode handlers, the effort and cost involved in performing an attack could increase significantly. In the rest of this paper, we'll describe in detail how to construct such a scheme.

V. OVERVIEW

CoDiver consists of four components as follows.

A. VIRTUAL INSTRUCTION SET AND HANDLERS

We design a set of virtual instructions and handlers to translate the virtual instructions to native code. In this work, we target the x86 instruction set. Our virtual instruction set is based on a stack machine architecture and is Turing-equivalent to the native machine code. The virtual instruction set design will be discussed in Section VI.

B. NATIVE CODE TRANSLATION

We develop a tool to convert native machine code into virtual instructions and then store them as bytecode automatically. Details of this tool are described in Section VII-A.

C. BYTECODE DIVERSIFICATION

We use a special encoding scheme to diversify the generated bytecode instructions, and divide each protected code region into several partitions. In each partition, we mapping the opcodes of the virtual instructions to different native code.

It means, even if an attacker finds a mapping rule from opcode to native code in one segment, the rule is difficult to be applied to other segments. We also obfuscate each handler to produce a set of obfuscated handlers where handlers follow different execution paths at runtime but produce identical results for a given virtual instruction. We also protect the core component of the VM using anti-taint analysis. This is a critical component of CoDiver, described in Section VII-B.

D. PE REFACTORING

Finally, the generated bytecode program and other VM components will be linked together with binary rewriting.

VI. VIRTUAL INSTRUCTION SET AND HANDLERS

In any code obfuscation system that based on VM, virtual instruction set and handler are the key foundations. The virtual instruction set must be Turing-equivalent to target native machine code, which means that any native instructions could replace the virtual instructions. Virtual instructions ultimately will be executed by the hand-crafted `handlers`, and these `handlers` are written in native instructions.

There are two main VM architectures: stack-based, and register-based. Examples of stack-based virtual machines are the Java Virtual Machine and the.Net CLR, and examples of register-based virtual machines are the Lua VM and the Dalvik VM. Here, we choose the stack-based architecture for our VM-based obfuscation system, the reasons are as follows:

- In stack-based VM, operations are performed with the help of the stack, which stores the operands and the results of operations. This could simplify the addressing of operands and the implementation of `handlers` ultimately.
- Native x86 instructions can be converted to virtual instructions more easily.
- During a given computation, stack-based VMs need more virtual instructions. This would make the instructions become more complex and precisely in line with our goal of preventing reverse analysis.

To devise a virtual IS that is Turing-equivalent to the native IS, one naive approach is devising a virtual instruction for every single native instruction. However, this results in a very large size of handlers. As the basic idea of stack-based VMs implies, a native operation is carried out or virtualized by virtual instructions in a three-phase fashion: pushing operands into the stack, executing the aimed operation, and storing the result into the virtual context. Therefore, it is sufficient for the virtual IS to include the following instructions:

- `load` instructions and `store` instructions are used for data transmission. `load` instructions are used to push operands onto the stack, and `store` instructions are used to pop the results out of the stack and store them into the virtual context.
- Arithmetical and logical instructions. Variants of these instructions are much smaller than their native ones, because the addressing mode of operands is simpler and more uniform, i.e. stack-based addressing.
- Branch instructions used to change the control flow of the bytecode program.

Other instructions not included in the above categories are defined as special virtual instruction - `undef`, which we will discuss later. We first discuss the different formats of these instructions and how to implement the handlers of them.

A. LOAD AND STORE INSTRUCTIONS

`load` and `store` instructions are used for preparing operands and storing the results of operations. They are used in the first and third phases of virtualizing a native instruction. In our virtual IS, they are the only ones that have operands. For a `load` instruction, the operand could be a virtual register, a memory addresses, or an immediate value, and for a `store` instruction, the operand is a virtual register or a memory address. Virtual registers are stored in the virtual context, i.e., the `VMcontext`. A naive construction of `VMcontext` is simply copying the values of native registers into the `VMcontext`. But the mapping between the virtual registers and the native registers is not necessarily one-to-one. To further impede the reverse analysis, the mapping mechanism could be made purposely more complex, as NISLVMP [11] does. In this section, we only consider the one-to-one mapping between the virtual registers and the native ones.

Besides the operand type, the operand size matters as well. In x86 architecture, the size of an operand could be 8-bit, 16-bit, and 32-bit. For example, given a memory address, the `load` instruction could fetch the first 8-bit, or the lower 16-bit value, or the entire 32-bit value that stored in that address. Therefore, it is better to design a virtual instruction for every distinct combination of operand type and size. However, in x86 architecture, `push` and `pop` operations do not support 8-bit operations. We decide to delay distinguishing different size operands to the second phase of virtualizing native instructions. Table 1 shows the virtual instructions of `load` and `store` and their corresponding `handlers`. In table 1, there exist four special virtual instructions: `load_r8h` and `store_r8h` are used when we encounter an operation that manipulate the second least significant byte of a register, i.e., `ah`, `dh`, `ch`, `bh`, `load_ms` and `store_ms` have no operands and are used to process instructions with indirect memory addressing mode (memory address is stored in a register or presented as an expression). An example in table 5 illustrates the situation of using of `load_ms`.

B. ARITHMETICAL AND LOGICAL INSTRUCTIONS

Arithmetical and logical virtual instructions are used to execute the aimed operations and they do not need to worry about operands, since their operands have been pushed into the stack by `load` instructions. However, as we said before, these instructions must consider the size of the operands. These instructions are in similar forms and we take `add`

TABLE 1. The virtual instructions and corresponding handlers of `load` and `store`.

VI	Handler

`load_r reg`	`movzx eax, byte [VPC]` *;get vr index*
	`add VPC, 1`
	`push dword [VMcontext+eax*4]`

`load_r8h reg`	`movzx eax, byte [VPC]` *;get vr index*
	`add VPC, 1`
	`movzx eax, byte [VMcontext+eax*4+1]`
	`push eax`

`load_m mem`	`mov eax, dword [VPC]` *;get memory addr*
	`add VPC, 4`
	`push dword [eax]`

`load_i8 imm8`	`movzx eax, byte [VPC]` *;get 8-bit imm value*
	`add VPC, 1`
	`push eax`

`load_i16 imm16`	`movzx eax, word [VPC]` *;get 16-bit imm value*
	`add VPC, 2`
	`push eax`

`load_i32 imm32`	`mov eax, dword [VPC]` *;get 32-bit imm value*
	`add VPC, 4`
	`push eax`

`load_ms`	`pop eax` `push dword [eax]`

`store_r reg`	`movzx eax, byte [VPC]` *;get vr index*
	`add VPC, 1`
	`pop dword [VMcontext+eax*4]`

`store_r8h reg`	`movzx eax, byte [VPC]` *;get vr index*
	`add VPC, 1`
	`pop edx`
	`mov byte [VMcontext+eax*4+1], dl`

`store_m mem`	`mov eax, dword [VPC]` *;get memory addr*
	`add VPC, 4`
	`pop dword [eax]`

`store_ms`	`pop eax`
	`pop ebx`
	`mov dword [eax], ebx`

Note: In the table, `reg` means register, `mem` memory address, `imm` immediate value, and `vr` virtual register. `VPC` is short for Virtual Program Counter and represents the address of the next bytecode instruction to interpret.

operation as an example to illustrate. Table 2 lists the virtual instructions and handling procedures of `add` operation. Since the operations of these instructions could be 8-bit, 16-bit, or 32-bit, we design virtual instruction for each of the operations of different operand sizes.

TABLE 2. The virtual instructions and handlers of `add` virtual instructions.

VI	Handler
`add8`	`pop eax` `add` *byte* `[esp], al`
`add16`	`pop eax` `add` *word* `[esp], ax`
`add32`	`pop eax` `add` *dword* `[esp], eax`

C. BRANCH INSTRUCTIONS

In native IS, the commonly used branch instructions include `jmp`, `jcc` (conditional jump), `call` and `retn`. It is a big challenge to virtualize these instructions, since they have many different variants and each of these variants needs a virtual instruction. Considering the destination of a jump (branch) instruction, if its destination also resides in the critical code, we call it an *inner jump*; otherwise, we call it an *outer jump*. Figure 3 illustrates these two kinds of branch instructions. In CoDiver, we ignore the situation where the destination of a branch instruction outside the critical code resides in the critical code, the start instruction of the critical code excluded. This situation will not occur if the critical code to be protected is well-structured.

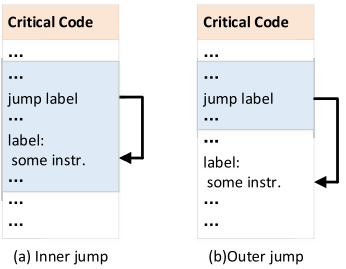


FIGURE 3. The destination of a branch instruction could be in the critical code or outside the critical code. Branch instructions include `jmp`, `jcc`, `call`, and `retn`.

Besides the location of the destination instruction, we should also consider if the destination of a branch instruction can be determined statically. From this point of view, the branch instructions can be divided into two categories: one is *direct branches*, whose destinations are calculated in a PC (Program Counter)-relative mode and can be calculated statically. The other is *indirect branches*, whose destinations are stored in registers or memories. Their destinations are undefined statically and are determined at runtime. Table 3 classifies different forms of `jmp/jcc/call/retn` instructions considering the above two categories.

TABLE 3. Examples of direct and indirect branches.

Direct Branches	`jmp rel8/16/32`
	`jcc rel8/16/32`
	`call rel16/32`

Indirect Branches	`jmp reg32/mem32`
	`call reg32/mem32`
	`retn`

Note: In this table, `rel` means a PC-relative address. The numbers (8, 16, and 32) are the size (in bit) of the operands.

1) DIRECT BRANCHES

Since `jmp` instructions are the basis of the other branch instructions, we elaborate on the introduction of the virtual instructions and handlers of `jmp` instructions. As we have illustrated, the destination of a direct `jmp` instruction can be calculated statically. If the destination instruction of the `jmp` instruction resides in the critical code, the `jmp` is a *direct inner* `jmp`; otherwise a *direct outer* `jmp`. We designate virtual instructions for both of them, `jmp_di` for the former, and `jmp_do` for the latter. For *direct inner* `jmp`, it is able to obtain the corresponding bytecode instruction address of its native destination instruction during protection. Therefore, we set that bytecode instruction address as the operand of `jmp_di`, which is pushed into stack by `load_i`. The handler of `jmp_di` fetches the address from stack and assigns it to VPC (Virtual Program Counter). For *direct outer* `jmp`, we just need to jump to that destination instruction.

Prior to that, we should restore the native context. Therefore, the operand of `jmp_do` is the address of the native destination instruction. Table 4 presents the above two virtual instructions of *direct* `jmp` and their handlers.

TABLE 4. The virtual instructions and handlers of `jmp` instructions.

VI	Handler
`jmp_di`	*;operand: addr of the dest. bytecode instr.* `pop eax` *;get operand* `mov VPC, eax`
`jmp_do`	*;operand: addr of the dest. native instr.* `pop [mem]` *;get operand* `...` *;restore native context* `jmp dword [mem]`

Note: `jmp_di` is for *direct inner* `jmp` and `jmp_do` for *direct outer* `jmp`.

`jcc` and *direct* `call` instructions are similar to *direct* `jmp`. `jcc` has many different kinds of instructions for different conditions, and each needs a specific virtual instruction. In the `handlers`, some extra instructions are needed to check the state of the conditions and decide to jump or not. `call` instructions can be considered as a `push` instruction followed by a `jmp` instruction. The `push` instruction pushes the return address into the stack and the `jmp` instruction jumps to the address of the subroutine.

2) INDIRECT BRANCHES

Since it is difficult to obtain the address of an indirect branch, we cannot decide whether the branch is an inner one or an outer one. One solution is to delay the decision until runtime. In such a case, however, the implementation of the handler is complex. Hence, for these instructions, instead of using a similar idea as that for *direct branches*, we use a special virtual instruction `undef`, which will be introduced later.

D. OTHER INSTRUCTIONS

The above three categories cover the commonly used instructions. Although the other native instructions are rarely used, such as `bts`, `enter`, `int n`, and `out`, our native IS should consider them too. For these instructions, we define a special virtual instruction - `undef`. At runtime, when encountering such an instruction, it first restores the native context and exits the VM. Then, it executes that native instruction in the native context and finally re-enters the VM and continues to execute the left bytecode instructions. The *indirect branches* are also processed in this way.

VII. OFFLINE CODE OBFUSCATION

In this section, we describe how to convert native instructions into virtual instructions and store them in bytecode format.

A. NATIVE INSTRUCTIONS TO VIRTUAL INSTRUCTIONS

In the obfuscation, we first convert native instructions into virtual instructions. This process has three steps: first, use `load` virtual instructions to load the operands into the stack; then, perform the target operation; finally, use `store` virtual instructions to put the result into a virtual context or a certain memory address. Table 5 gives some examples of native instructions and their corresponding virtual instructions.

TABLE 5. Examples of native instructions and their corresponding virtual instructions.

Native Instr.	VI
`mov eax, ebx`	`load_r 3` `store_r 0`

`mov eax, dword [esi+4]`	`load_r 6`
	`load_i32 4`
	`add32`
	`load_ms`
	`store_r 0`

`add eax, edx`	`load_r 0`
	`load_r 2`
	`add32`
	`store_r 0`

`jmp 4020a8h` (direct inner jump)	`load_i32 42a583h` `jmp_di`

Note: `42a583h` is the bytecode instruction address that corresponds to the native instruction at `4020a8h`.

Data transfer instructions are mainly mapped to `load` and `store` instructions. Typical examples of these instructions are `pop`, `mov`, and `push`. Arithmetical and logical instructions will follow the three-phase processing strictly. Branch instructions will be mapped to a `load` instruction, and followed by the branch virtual instructions. The native instructions which has complex addressing mode will use the above virtual instructions for iteratively processing, for example, the "`mov eax, dword[esi+4]`" instruction in Table 5.

B. VIRTUAL INSTRUCTIONS TO BYTECODES

Virtual instructions will be encoded into bytecodes in the end. It is similar to that an assembler assembles assembly instructions into machine code and only can be interpreted by the virtual interpreter of a VM-based protection system. We adopt an encoding scheme less compacted than the x86 instruction architecture which uses separate bytes for the *opcode* and *operand* of virtual instruction. In practice, we assign a distinct ID to each virtual instruction as its *opcode*. Using this ID as an index, `VMloop` looks for the address of the virtual instruction `handler` in the address table that records the address of each `handler`. The number of virtual instructions is under 256, so one byte is enough to encode their IDs. For *operands*, because they could be different in size,[1] so here we use one, two, or four bytes to encode them accordingly. Figure 4 shows examples of virtual instructions and their bytecode. The figure also demonstrates how `VMloop` fetches and interprets the bytecode instructions.

C. RANDOMIZE THE SEMANTICS OF BYTECODE INSTRUCTIONS

We can see from the demo above, if the attacker knows the semantics of a bytecode instruction, when he encounters it

[1]The *operand* of a virtual instruction may be the index of virtual register, an immediate value, or a memory address. So their size could be varied: a virtual register index is 8 bits, an immediate value is 8/16/32 bits, and a memory address is 32 bits.

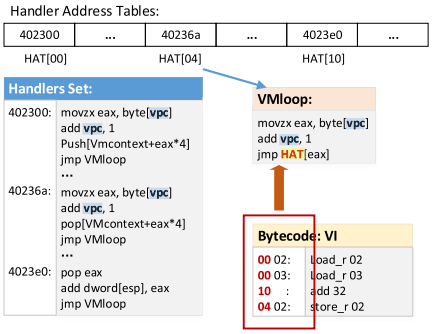


FIGURE 4. Examples of virtual instructions and their bytecode. Each virtual instruction is encoded as a bytecode instruction that contains an *opcode* and an optionally *operand*. The bytecode instructions is passed into `VMloop`, which uses the *opcode* of each bytecode instruction as an index to find the address of the corresponding handler in the HAT (Handler Address Table).

next time, he doesn't need to re-analyze its `handler` to find out what it could do.[2] As shown in Figure 4, by analyzing `Handler_4023e0`, an attacker could find out the bytecode instruction "`10`" presents an addition operation. Then the next time he encounters this bytecode instruction "`10`", he will know immediately that it performs an addition operation.

In order to make previously obtained *analysis knowledge* difficult to be reused, we need to break the fixed correspondence between virtual instructions and semantics and randomize them. We can achieve this goal base on the encoding scheme. The main idea of this scheme is to change the relationship between the IDs (*opcodes*) and the virtual instructions, which is similar to [17]. Each time the virtual instructions are encoded, the IDs will be shuffled first. Then the virtual instructions are encoded with scrambled IDs. The addresses of `handlers` are also stored in the handler address table accordingly.

D. PARTITION BYTECODE PROGRAM

After randomized the semantics of bytecode instructions, an attacker can not reuse this *analysis knowledge* to figure out the real function of a bytecode instruction. However, there is still a way to bypass the effect of randomization. As shown in table 5, every virtual instruction has its own frequency, we can see `load_r` and `store_r` are two of the most frequent virtual instruction that been used. So even after randomization, an attacker still could find out the semantics of bytecode instructions according to the probability distribution of *opcodes*. This problem could come down to the static mapping relationship between IDs and virtual instructions in each use. Although we have executed randomization, the probability distribution property of the virtual instruction will still be passed to the ID been used this time.

[2]Because programmer could modify `handlers` to hinder analysis, it saves the analyst a lot of time without having to bother to analyze them.

1) BYTECODE PROGRAM PARTITION DESIGN

To frustrate the inferences based on the frequency statistic of *opcodes*, we need to break the static mapping relationship. So all the generated virtual instructions are partitioned into several parts, and each part is encoded differently. Especially, instead of encoding all the generated virtual instructions simultaneously during obfuscation, we encode every part separately. Before an encoding process is executed, we first shuffle the IDs of virtual instruction randomly, then encode it with the result. With shuffles, an identical *opcode* in different parts of the bytecode program may reveal different semantics, thus the probability distribution property of *opcodes* are obscured. As shown in Figure 5, partition the virtual instructions into two parts and encode each part of virtual instruction into different opcode. For example, in the first part `load_r` is encoded into "`00`", yet in another part is "`7a`".

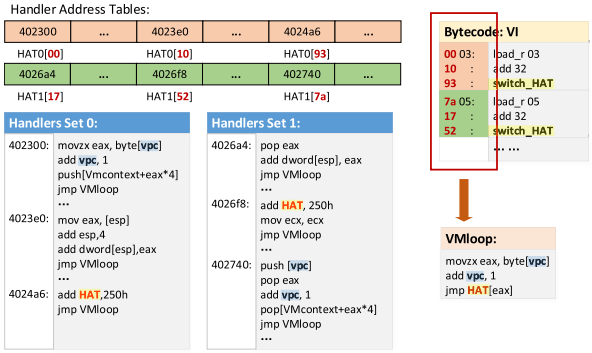


FIGURE 5. Example of partition, and there are two parts in this figure. In different part, the virtual instructions are encoded differently and interpreted with different `handlers` set. So the number of HAT increases accordingly. To switch the currently used (by `VMloop`) HAT to the next one, we add a new virtual instruction `switch_HAT`. The *operand* of `switch_HAT` is the size of a HAT.

Algorithm 1 Partition Bytecode Program

Require: VIs of critical code segment, partition number N.
Ensure: Bytecode program with N partition.

```
Apply memory space M;
VIs randomly divided into N partitions;
Generate N sets of handlers by using algorithm 2;
while N ≠ 0 do
    Take a partition;
    Randomly select a set of handlers and shuffle the
    IDs of virtual instructions;
    Build a corresponding HAT;
    Use the shuffle results to encode bytecode program P;
    Store the P to M;
    N decrement operation
    N − −;
end while
```

Algorithm 1 demonstrates partition encoding. To partition the bytecode program, we randomly divide the virtual instructions into N partitions and obfuscate the original HAS

(Handler Set) N times to get N sets of handlers, and the detailed HAS obfuscation approach are demonstrated in section VII-E. For each partition, we will randomly select one set of handlers and shuffle the IDs of virtual IS, and then use the results to encode the VIs into a specific bytecode program. In the end, a complete bytecode program is generated, which has multiple partitions and the same bytecode has different semantics in different partitions. These specific bytecodes only can be interpreted by the system's virtual interpreter.

For the VMloop uses the *opcodes* of bytecode instructions as the indexes of their corresponding handler addresses, and each part will be encoded differently, so every part needs his own HAT. At the end of each part, the HAT used by it will switch to the HAT of the next part. The new virtual instruction - switch_HAT This mission is executed by a. Because switch_HAT is always added to the end of a part, and the order of HATs is in accordance with partitions. So we need to add the size of a HAT to the HAT pointer that been used by VMloop (as shown in figure 5 that Handler_4024a6 does). In our prototype, the number of Handlers is 148 and the address of a Handler is 4 bytes, so the size of a HAT should be 592 (250h in hexadecimal) bytes.

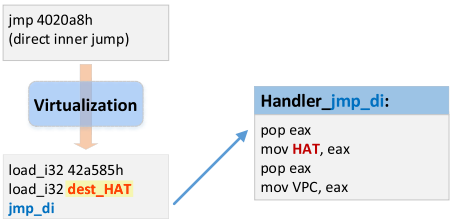


FIGURE 6. The virtualization of a *direct inner* transfer instruction with HAT switching. Instruction load_i push the address of the destination HAT into stack, and assign it to the HAT pointer that is used by VMloop at runtime.

HATs switching is not only limited to the end of every part. Switching can also occur when the target of a branch instruction is in a different part. We know, a branch instruction could change the control flow of a program though changing the VPC. When encounter such an instruction, we cannot just only add a switch_HAT to it, because if we change the VPC to a location in another part, switch_HAT might not be interpreted by VMloop. So we put the code for the switch in the Handlers of the branch instruction. In this case, the branch instructions indicating the *direct inner*, as *direct outer* branches and *indirect* branches will all leave the virtual context and we do not need to worry about the HATs switching. In the process of protection, we will first calculate its destination for each *direct inner* branch instruction, and then calculate the partition where the destination should reside. The HAT address of the partition is pushed onto the stack by load_i and will be used by the Handler of the branch instruction to set the value of the HAT pointer used by VMloop. Figure 6 shows the virtualization of *direct inner* branch instruction, as compared to that in table 4.

2) SECURITY ANALYSIS OF PARTITION DESIGN

Assume there is an attacker now, which uses the attack method based on virtual execution that introduced in section III to reverse the bytecode partition program. First, the attacker needs to perform dynamic debugging the target program, and then spend some time to locate the address of VMloop from the obfuscated virtual interpreter. Next, he needs to collect the bytecode program by analyzing the parameters of VMloop.

To restore the logical function of the original code, the attacker needs to analyze the semantics of these extracted bytecodes. But for the target program, which generated by using special encoding schemes and its bytecode program is divided into multiple partitions. So the attacker must first obtain the partition information of critical code to further reverse analysis, but it is difficult for him. After the NIs have been converted to a VIs, there is a HAT switching operation by using VI - switch_HAT at the end of each partition. However, virtual instruction is just an intermediate language in the process of protection, and it does not appear in the final program. All of the instructions will eventually be in the form of bytecodes, and the bytecode semantics of each partition is randomly changed (such as bytecode "93" and "52" in figure 5). So the attacker cannot use this feature to delineate the partition. He can only spend a lot of time through continuous tracking and debugging to predict the partition of the transformation.

Assume that the attacker gets the partition information after a lot of analysis. Next, the attacker needs to analyze the bytecode of each partition and extract the semantic information implied by the handlers. Because the bytecode of each partition has different semantics, and their mapping handlers are also with different forms. Therefore, the attacker cannot reuse the previous *analysis knowledge* to attack the next partition, and for different target program that protected by CoDiver is more like this. Unique partition configuration and different handler sets so that the attacker cannot achieve the batch automatic attack by building an attack knowledge base. The attacker has to spend a lot of time to analyze every detail for each program.

E. OBFUSCATE THE HANDLER SETS

To further impede automated reverse analysis, we can obfuscate the handlers, and use different obfuscation strategies for handlers in different partitions. As a result, a handler in different partitions will look different. An analyst cannot immediately recognize them as the same one and needs to analyze each of them, which increases the workload of the analyst. At the same time, it can also prevent the attacker use attack knowledge base to match these handlers for automatic reverse analysis.

Our obfuscation method is shown in Algorithm 2. The number of HAS obfuscation will be determined by the number of partitions. Our system will select several methods from the obfuscation library randomly. This library contains

Algorithm 2 Virtual Interpreter Obfuscation

Require: The original HAS, partition number N.
Ensure: N equivalent HASs with different forms.

```
while N ≠ 0 do
    Apply memory space M1;
    Take the first handler from HAS;
    while There are still untreated handler do
        Randomly select multiple obfuscation methods and
        using order of them;
        Obfuscate handler and store the results to M1;
        Take the next handler from HAS;
    end while
    Apply memory space M2;
    Using the anti-taint analysis technique to protect the
    code in M1;
    Store the results to M2, and release the memory space
    of M1;
    N decrement operation
    N − −;
end while
```

equivalent instruction substitution [27], control flow flattening [29], code out-of-order [28] and junk instructions injection. Then the system will obfuscate the `handler` by these selected methods. Finally, we have multiple equivalent HASs but with different forms. After obfuscation, we will still use some anti-taint analysis techniques (more details are presented in section VII-F) to protect the HASs. This approach could protect the virtual interpreter from the attack based on de-obfuscation effectively.

For instance, HAS is an original `handler` set consist of m `handlers`. HAT stores the addresses of these `handlers`, whose indexes correspond to the *opcode* of the virtual instruction. We first use different strategies to obfuscate HAS for n times, and n is determined by the number of partitions. Then we can obtain multiple HASs with different forms but they are semantic equivalence. At this point, every equivalent `handler` still has the same index, this means the relationship between them is direct mapping and it is insecure. Therefore, as mentioned in prior section VII-D, the method of bytecode program partitioning and semantics of bytecode instructions randomization. We will first shuffle the virtual instruction IDs randomly, and then provide a new HAT for every partition (as shown in figure 5) with these results. With this shuffle method, in different parts of the bytecode program, the same *opcode* may correspond to different semantics. So in different HASs, the relationship between these equivalent `handlers` could be:

$$\mathrm{HAS}_1(i) \Leftrightarrow \mathrm{HAS}_2(j) \Leftrightarrow \ldots \Leftrightarrow \mathrm{HAS}_n(k), \quad 1 \le i, j, k \le m.$$

We can see, with these different forms of `handlers` and various semantics of bytecode instructions, even though the attacker has rich knowledge of attack, it is still difficult to reuse the knowledge to perform automatic reverse analysis. Attackers have to start from scratch and spend a lot of time in detail analyzing.

F. ANTI-TAINT ANALYSIS

Simply obfuscating the `handlers`, however, cannot prevent an attacker from reverse engineering completely. There are several existing de-obfuscating techniques that can be used to counter the traditional virtualization protection. Such as Yadegari *et al.* [25] use equational reasoning about assembly-level instruction semantics to simplify away obfuscation code from execution traces of emulation-obfuscated programs. Tang *et al.* [26] use taint propagation to track the value flows from the inputs of the programs to the outputs, and simplify the logic of the instructions that operate on and transform values through this flow by semantics-preserving code transformations.

However, the work of Coogan *et al.* is working on the base of equational reasoning about assembly level instruction semantics. And simplify these complex equations are difficult, so it is also hard to separate the complex control flow or different components of nested loops. The work of Yadegari *et al.* is effective even applied to previously unseen obfuscations, but the code simplification should be first to identify input-to-output data flows. This relies on the taint propagation and analysis to identify the explicit value flows from inputs to outputs, and then identify implicit flows by control dependence analysis. Therefore, these methods could be obstructed by enforcing dataflow obfuscation to the handling procedures [20]. We adopt some anti-taint analysis techniques to protect the data flow of the `handlers` from taint analysis. Specific ways are as follows:

a: THE TRANSFER OF NAIVE MODEL

A simple case is given in figure 7-(a), assuming that there is a variable B that has been marked by taint, we need to transfer the value of B to `AX`. If we use the `MOV` instruction, `AX` will also be tainted. The transfer of naive model is that to do subtraction operation for B, at the same time to do addition operation for `AX`. When the value of B is reduced to "0", the value of AX increased from "0" to B, and `AX` will not be marked as a tainted data.

b: THE TRANSFER OF CONTROL DEPENDENCIES

Control dependence analysis is an important step towards the process of taint analysis. For a set of tainted data, to carry out anti-taint analysis processing. As shown in figure 7-(b), we can launder tainted data by assigning the data that no tainted directly to the tainted data. This process needs to match the tainted data and not tainted data, in order to ensure the correctness of the data will not be affected.

c: THE INDIRECT TRANSFER OF STACK POINTER

As we can see from figure 7-(c). The indirect transfer of stack pointer took advantage of the working principle of the stack to implement the anti-taint analysis. The principle of this method is to first put a set of no tainted data into the

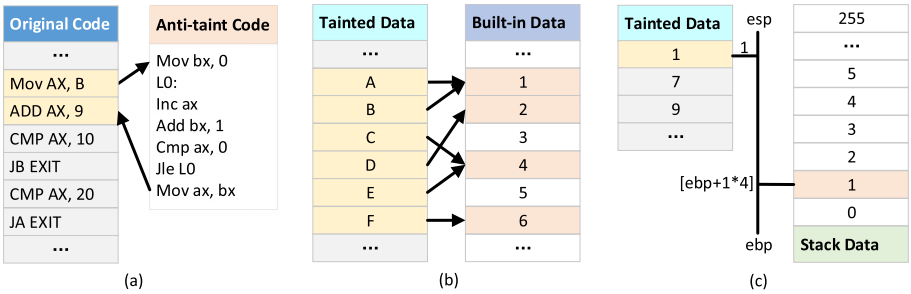


FIGURE 7. Examples of three anti-taint analysis techniques. Respectively as, (a) the transfer of naive model, (b) the transfer of control dependencies and (c) the indirect transfer of stack pointer.

stack and then through the address pointer of stack data to access them, and finally use them to replace the tainted data to implement the anti-taint analysis.

These three methods could effectively prevent the dissemination of tainted data and resist the taint analysis by laundering tainted data.

VIII. EFFECTIVENESS EVALUATION

CoDiver could effectively block malicious reverse analysis by invalidating the existing *analysis knowledge*. We have learned in Section II, for VM-based obfuscation program reverse engineering, it is indispensable to understand the semantics of bytecode instructions. But in this kind of program, semantics have been encapsulated in `handlers`, and it is tedious and error-prone to extract them form `handlers`. Therefore, it would save the analyst a lot of time and effort, if the semantics of bytecode instructions could be accessed without the need to re-trace and re-analyze the `handlers`. The main goal of CoDiver is right to frustrate this attempt, and constrain the attacker to analyze the `handlers` each time. With the randomized process of the bytecode instruction semantics, the same instruction may represent different instances of obfuscation. Even in the same obfuscation program, with applying different encoding schemes to different partitions of the bytecode program, the result may still different, which could confuse the analysts and increase their workload largely.

As assuming the number of virtual instructions, `handlers`, is H, then in two different obfuscated programs, the probability of a bytecode instruction has the same semantics is $\frac{1}{H}$. The total shuffling time of *opcodes* is $H!$. As in an obfuscation program, we assume the partition number of it is N, the probability of a bytecode instruction in different partition having the same semantic is $\left(\frac{1}{H}\right)^{N-1}$. So we believe that CoDiver could invalidate the *analysis knowledge* of bytecode instruction semantics effectively.

The average frequency of each *opcode* been used in the benchmark is also calculated. We use the obfuscation programs to represent different partition numbers like 2, 8, and 32 for comparison. As shown in figure 8, with the increase of the partition number, the frequency of different *opcodes* become more similar. In contrast, when the number

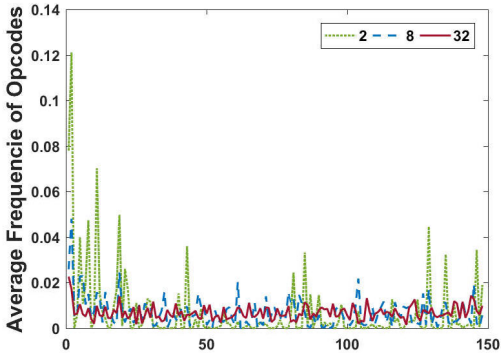


FIGURE 8. The average frequency of using an *opcode* during execution. The graph displays opcode ID on the horizontal axis and the frequency (normalized between 0 and 1) of each opcode to be used on the vertical axis.

of partitions is very small, it is easy to see that the most common instruction "`load_r`" has the highest frequency.

In order to evaluate our method's spatial and temporal overhead, we implemented a prototype of CoDiver on the Windows platform to obfuscates x86 PE executable programs. In implementation, we designed `148` virtual instructions and the corresponding `handlers`. All experiments were performed on a Dell Optiplex 960h using Intel®Core™ 2 Duo Processor E8400 at 3.00 GHz with 4.00 GB of RAM. The operating system was Windows 7 Enterprise.

We use CoDiver to protect four x86 PE executable programs, as `md5`,[3] `gzip`,[4] `bcrypt`,[5] `mat_mul`.[6] We use the first three programs to process a 10KB text file (`test.txt`), and use `mat_mul` to calculate the product of two 5×5 matrices. The statistics of these executables are shown in table 6. We select a key piece of code from each program to protect. This protection process will be performed for 10 times, during each execution we specifying a number (from 1 to 10) as the parameter of partitions.

Figure 9 shows the change of code size under the action of the obfuscated program. The graph displays the number of partitions that been used in the obfuscated method on the

[3]MD5. http://www.fourmilab.ch/md5/.
[4]gzip. http://www.gzip.org/#sources.
[5]Bcrypt - Blowfish file encryption. http://sourceforge.net/projects/bcrypt/.
[6]Matrix Multiply. https://github.com/MartinThoma/matrix-multiplication.

TABLE 6. Statistics of the target programs.

Target Program	Version	Size(KB)	Critical Code	N1	N2	Ratio	N3
md5.exe	2.3	11	Transform()	1327	563	42.36%	85013
gzip.exe	1.2.4	56	deflate()	10181	153	1.50%	539082
bcrypt.exe	1.1.2	68	Blowfish_Encrypt()	2997	54	1.80%	1735710
mat_mul.exe	-	184	ijkalgorithm()	49327	60	0.12%	84325

Note: Columns 5 and 6 gives the entire program's instruction count and the critical functions, and column 7 gives the proportion of key code. We use Pin [30] to count the number of instructions dynamically executed in the key function, which is shown in the last column. We only select 60 special instructions of `mat_mul` to verify the impact of CoDiver on program overhead while protecting code. The results are shown in figure 10, with no obvious influence.

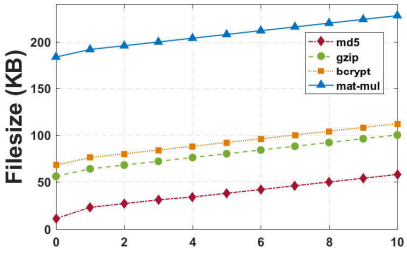


FIGURE 9. The impact of CoDiver on code size (KB). The file size slightly increased with the increase of number of partitions.

horizontal axis, and the number "`0`" for the original program. While the number of partitions increases, so does the number of bytes in the final program, and these increments are mainly come from the increase of HATs and HAS. However, for the size of the HAT is just 592 bytes, so it grows slowly. Also, the PE executable file is aligned to a value (4096 or 512 usually) [31], so the final size of the program is mainly determined by the number of HAS, which increases regularly with the grows of the partition number.

To estimate the runtime overhead of CoDiver, we will run the obfuscated programs multiple times and then take the average of the execution times, the results are shown in figure 10. Amongst these, the execution time of `bcrypt` increases by the most than original program.[7] For the key instructions in `bcrypt` is frequently executed than others (as shown in the bottom row of the table 6). In addition, execution time does not change much with the increase of partition number. In section VII-D, we can see that the addition of each one part, the obfuscated program only needs to execute one more `handler` to interpret the `switch_HAT` instruction. The resulting runtime overhead could be negligible.

We estimate the average runtime overhead of each dynamically executed instruction. T_{ob} is used to represent the execution time of a obfuscated program, T_o for the execution time of the original program, and C_e for the number of key instructions executed dynamically. Calculate the average runtime overhead for each dynamically executed instruction according to

$$(T_{ob} - T_o)/C_e.$$

Figure 11 shows the results. We can see that `md5` has the highest average runtime overhead. This is because the key code of `md5` contains many arithmetical and logical instructions, so it tends to take longer to interpret.

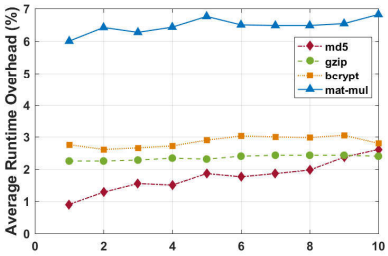


FIGURE 11. The average runtime overhead per dynamically executed critical instruction.

Finally, we compare CoDiver with two commercial code virtualization protection systems, VMProtect [9] and Themida [10]. We select some instructions from the previous four test cases, using CoDiver, VMProtect, and Themida to protect them respectively. Next, the size and average execution time of these protected programs are compared. Figure 12 shows the effects of these three virtualization protection systems on the file sizes of these four test programs.

[7]The execution time of the original program is specified by "`0`" on the horizontal axis.

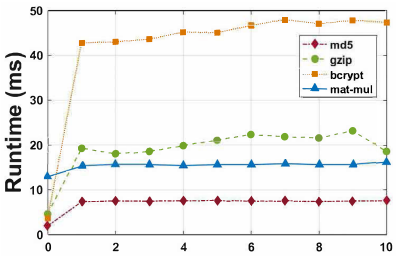


FIGURE 10. The impact of CoDiver on runtime performance (μs) with different partitions.

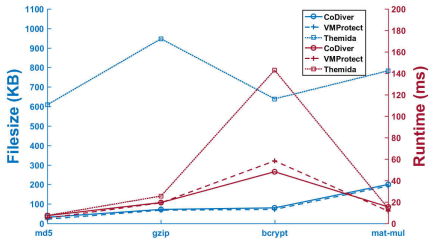


FIGURE 12. The comparison of impact on file size (KB) and runtime performance (μs) with VMProtect and Themida.

CoDiver and VMProtect are comparable in file size and less than Themida. This result is related to the design of the virtual instructions and `handlers`. The runtime overheads of all systems are shown in figure 12. Overall, the costs of these three protection systems are similar. In particular, the runtime overhead of Themida protecting `bcrypt` is much greater than the other target programs. This result may be related to Themida's design and the number of key instructions executed in `bcrypt`. From the above comparison, we can see that CoDiver and VMProtect are similar in time and space overhead, and both of them are better than Themida.

IX. RELATED WORK

In recent years, people have proposed the code deobfuscation technique. Yadegari *et al.* [25] proposed a method to identify the instructions related to system calls, and extract the approximate dynamic tracking of the original code automatically. Tang *et al.* [26] proposed a method to track the flow of input values and then simplify the logic of the instructions by semantic preserving code transformation. However, these methods can only extract certain execution characteristics of the target program, but can not completely restore the structure of the original code. Or they need taint analysis to track and analyze the data flow, which may be hindered by enforcing dataflow obfuscation to the handling procedures [20]. These methods are more or less limited.

CoDiver adopts ISR (Instruction Set Randomization) technology to generate random and unique virtual instruction sets. ISR has been widely used to prevent code injection attack by randomizing the underlying system instructions [17], [32], [33]. In our approach, using a set of random keys to encrypt the instructions, and then decrypt them before being executed by the CPU. ISR is effective to protect the system against code injection attack, but cannot prevent reverse engineering attack. In our attack model, software programs are executed in a malicious host environment, and attackers can trace and record the decryption instructions for later analysis. When generating random virtual instruction sets, CoDiver adopts a method similar to ISR, by changing the relationship between the opcodes and the virtual instructions [17], but it never ''decrypts'' the virtual instructions back to the original instructions. Instead, CoDiver uses `handlers` to interpret the virtual instructions, and the `handlers` of virtual instructions are more complex than their corresponding native instructions. Besides, CoDiver uses multiple partitions in a single program and has different ISRs, making reverse analyses more tedious and difficult.

X. CONCLUSION

This paper present CoDiver, a code obfuscation scheme base on the virtual machine. CoDiver is designed to prevent attacks by reverse engineering the code base on the knowledge obtained from analyzing the programs that been protected by the same code obfuscation technique. The core of CoDiver lies in a novel strategy to diversify the obfuscating process. To achieve this goal, we first divide the protected code area into different partitions. Then the opcode of each virtual instruction in each partition is randomly mapped to different bytecode handlers. Therefore, the mapping between virtual instruction and native code is different in different code partition. This makes the program behavior unpredictable and makes it harder to reuse the knowledge obtained from analyzing other programs to attack the target application. We evaluated our approach on real-world applications and compared it to the most advanced VM-based code protection tools. Experimental results show that CoDiver provides stronger protection with comparable overhead.

REFERENCES

[1] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, ''Protecting software through obfuscation: Can it keep pace with progress in code analysis?'' *ACM Comput. Surv.*, vol. 49, no. 1, 2016, Art. no. 4.

[2] P. Ananth, D. Gupta, A. Sahai, and Y. Ishai, ''Optimizing obfuscation: Avoiding Barrington's theorem,'' in *Proc. ACM Conf. Comput. Commun. Secur.*, 2014, pp. 646–658, doi: 10.1145/2660267.2660342.

[3] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, ''Obfuscation resilient binary code reuse through trace-oriented programming,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2013, pp. 487–498.

[4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, ''Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection,'' *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1157–1177, Dec. 2017.

[5] M. Dalla Preda and R. Giacobazzi, ''Control code obfuscation by abstract interpretation,'' in *Proc. 3rd IEEE Int. Conf. Softw. Eng. Formal Methods (SEFM)*, Koblenz, Germany, Sep. 2005, pp. 301–310.

[6] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, ''Mimimorphism: A new approach to binary code obfuscation,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2010, pp. 536–546.

[7] Z. Tang, M. Li, G. Ye, S. Cao, M. Chen, X. Gong, D. Fang, and Z. Wang, ''*VMGuards*: A novel virtual machine based code protection system with VM security as the first class design concern,'' *Appl. Sci.*, vol. 8, no. 5, p. 771, 2018.

[8] *Oreans Technologies Code Virtualizer*. Accessed: May 2019. [Online]. Available: http://www.oreans.com/codevirtualizer.php

[9] *VMProtect Software Protection*. Accessed: Apr. 2019. [Online]. Available: http://vmpsoft.com/

[10] *Themida Overview*. Accessed: 2019. [Online]. Available: http://www.oreans.com/themida.php

[11] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, and Y. Gu, ''NISLVMP: Improved virtual machine-based software protection,'' in *Proc. 9th Int. Conf. Comput. Intell. Secur.*, Leshan, China, Dec. 2013, pp. 479–483.

[12] N. Falliere, ''Inside the jaws of trojan. Clampi,'' Symantec Corp., NortonLifeLock, Mountain View, CA, USA, Tech. Rep., 2009.

[13] R. Rolles, ''Unpacking virtualization obfuscators,'' in *Proc. 3rd USENIX Conf. Offensive Technol. (WOOT)*. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–7.

[14] H. Wang, D. Fang, G. Li, N. An, X. Chen, and Y. Gu, ''TDVMP: Improved virtual machine-based software protection with time diversity,'' in *Proc. ACM SIGPLAN Program Protection Reverse Eng. Workshop (PPREW)*, New York, NY, USA, 2014, Art. no. 4.

[15] F. Fang, Y. Wu, S. Wang, and Y. Huang, ''Multi-stage binary code obfuscation using improved virtual machine,'' in *Proc. 14th Int. Conf. Inf. Secur. (ISC)*. Berlin, Germany: Springer-Verlag, 2011, pp. 168–181.

[16] M. Yang and H. Liu-Sheng, ''Software protection scheme via nested virtual machine,'' *J. Chin. Comput. Syst.*, vol. 32, no. 2, pp. 237–241, 2011.

[17] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and W. Zheng, ''Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling,'' *Comput. Secur.*, vol. 74, pp. 202–220, May 2018.

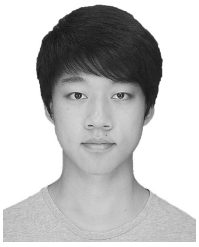
[18] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and H. Zhang, ''Exploit dynamic data flows to protect software against semantic attacks,'' in *Proc. UIC*, 2017, pp. 1–6.

[19] S. Chow, P. Eisen, P. C. van Oorschot, and H. Johnson, "A white-box DES implementation for DRM applications," in *Digital Rights Management*. Berlin, Germany: Springer, 2003, pp. 1–15.
[20] Y. Jia, T. Lin, and X. Lai, "A generic attack against white box implementation of block ciphers," in *Proc. Int. Conf. Comput., Inf. Telecommun. Syst. (CITS)*, Jul. 2016, pp. 1–5.
[21] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.
[22] *OllyDbg*. Accessed: Feb. 2019. [Online]. Available: http://www.ollydbg.de/
[23] *IDA Pro*. Accessed: Nov. 2018. [Online]. Available: https://www.hex-rays.com/index.shtml
[24] *Sysinternals Suite*. Accessed: Jun. 2019. [Online]. Available: https://technet.microsoft.com/enus/sysinternals/bb842062/
[25] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2015, pp. 674–691.
[26] Z. Tang, K. Kuang, L. Wang, C. Xue, X. Gong, X. Chen, D. Fang, J. Liu, and Z. Wang, "SEEAD: A semantic-based approach for automatic binary code de-obfuscation," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Sydney, NSW, Australia, Aug. 2017, pp. 261–268.
[27] N. George and G. Charalambous. *Applied Binary Code Obfuscation*. Accessed: Apr. 2019. [Online]. Available: http://www.intelligentexploit.com/articles/Applied-Binary-Code-Obfuscation.pdf
[28] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1144–1156, May 2014.
[29] B. Sandrine and T. Alix, "Formal verification of control-flow graph flattening," in *Proc. 5th ACM SIGPLAN Conf. Certified Programs Proofs (CPP)*, New York, NY, USA, 2016, pp. 176–187.
[30] *Pin-A Dynamic Binary Instrumentation Tool*. Accessed: May 2019. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binaryinstrumentation-tool
[31] *Peering Inside PE: A Tour of the Win32 Portable Executable File Format*. Accessed: Jun. 2019. [Online]. Available: https://msdn.microsoft.com/enus/magazine/ms809762.aspx
[32] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "Randomized instruction injection to counter power analysis attacks," *ACM Trans. Embedded Comput. Syst.*, vol. 11, no. 3, 2012, Art. no. 69.
[33] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proc. 26th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New York, NY, USA, 2010, pp. 41–48.

WEI WANG received the Ph.D. degree in information and communication engineering from Northwestern Polytechnical University, Xi'an, China. She is currently an Associate Professor with the School of Information Science and Technology, Northwest University. Her research interests include information security, wireless sensor networks, tour planning of mobile node, and cross technology communication.

MENG LI received the B.E. degree in computer science and technology from Northwest University, Xi'an, China, in 2017, where she is currently pursuing the M.S. degree in computer application technology with the School of Information Science and Technology. Her research interests include wireless sensing and information security.

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



HUANTING WANG received the B.E. degree in computer science and technology from Northwest University, Xi'an, China, in 2018, where he is currently pursuing the M.S. degree in computer application technology. His research interests include wireless sensing and information security.

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

FUWEI WANG was born in Xi'an, Shaanxi, China, in 1987. He received the Ph.D. degree from Xidian University, Xi'an, in 2014. He is currently working with Northwest University. His research interests include antenna scattering, software security, RCS reduction, and metamaterials.

JIE REN received the Ph.D. degree in computer science from Northwest University, in 2017. He is currently a Lecturer with the School of Computer Science, Shaanxi Normal University. His research interests include mobile computing, machine learning, and performance optimization.

XIAOQING GONG was born in Shanxi, China, in 1974. She received the B.S. degree in computer science, the M.S. degree in computer software, and the Ph.D. degree in computer software and theory from Northwest University, Xi'an, China, in 1995, 1998, and 2004, respectively. From 1998 to 2009, she was a Lecturer with the Software Engineering Institute, Northwest University, where she has been an Associate Professor with the School of Information Science and Technology, since 2009. She is the author of two books and more than a dozen articles. Her current research interests include software engineering and software security.

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

ZHENG WANG is currently an Associate Professor with the School of Computing, University of Leeds. He has published over 70 articles and received four best paper awards. His research focuses on compiler-based code optimization and systems security.