A

Enhance Code Virtualization Protections Via Diversity

Guanghui Li, Northwest University
Kaiyuan Kuang, Northwest University
Zhanyong Tang, Northwest University
Di Ma, University of Michigan-Dearborn
Dingyi Fang, Northwest University
Xiaojiang Chen, Northwest University
Zheng Wang, Lancaster University

Code virtualization built upon virtual machine (VM) technologies are emerging as a viable method for implementing code obfuscation to protect programs against unauthorized analysis. State-of-the-art VM-based protection approaches use a fixed set of virtual instructions and bytecode interpreters across programs. This, however, opens up a security hole where an experienced attacker can use knowledge extracted from other programs to quickly uncover the mapping between virtual instructions and native code for applications protected under the same scheme. In this paper, we propose a novel VM-code obfuscation system to address this problem. The core idea of our approach is to obfuscate the mapping between the opcodes of bytecode instructions and their semantics. We achieve this by partitioning each protected code region into multiple segments where the mapping of opcodes and their semantics is randomized in different ways in different segments. In this way, each bytecode instruction will be translated into different native code in different sections of the obfuscated code. This significantly increases the diversity of the program behavior. As a result, the knowledge of bytecode to native code mappings obtained from other programs is unlikely to be useful for a new program. We evaluate our approach on a set of real-world applications and compare it against two state-of-the-art VM-based code obfuscation approaches. Experimental results show that our simple approach is effective, which provides stronger protection at the cost of little extra overhead.

CCS Concepts: •**Security and privacy** → **Software security engineering;** *Software reverse engineering;*

Additional Key Words and Phrases: Virtualized obfuscation, reverse engineering, instruction set randomization, analysis knowledge

1. INTRODUCTION

Unauthorized code reverse engineering is a major concern for software developers. This technique is widely used by adversaries to perform various attacks, including removing copyright protection to obtain an illegal copy of the software, taking out advertisement from the application, or injecting malicious code into the program. By making the program harder to be traced and analyzed, code obfuscation is a viable means to protect software against unauthorized code modification [**?**; **?**; **?**; **?**; **?**; **?**].

Code virtualization based on a virtual machine (VM) is emerging as a promising way for implementing code obfuscation [**?**; **?**; **?**; **?**; **?**; **?**; **?**]. The underlying principal of VM-based protection is to replace the program instructions with bespoke virtual instructions and eventually encode into a bytecode program. These virtual bytecodes will then be translated into native machine code at runtime to execute on the under-

Author's addresses: G. Li, K. Kuang, Z. Tang, D. Fang, X. Chen, School of Information Science and Technology, Northwest University, Xian, China, 710127; D. Ma, Department of Computer and Information Science, University of Michigan-Dearborn, MI, 48128; Z. Wang, School of Computing and Communications, Lancaster University, UK.

DOI: 0000001.0000001

lying hardware platform. This forces the attacker to move from a familiar instruction set to an unfamiliar environment, which can significantly increase the time and effort involved in the attack.

Reverse engineering of VM-obfuscated code typically follows a number of steps. The attack first reverse-engineers the virtual interpreter to understand the semantics of individual bytecode instructions, i.e. how will a bytecode map be translated to native code. The attacker then translates the bytecode back to native machine instructions or even high-level program languages to understand the program logic [**?**; **?**]. Among these steps, understanding the semantics of individual bytecode instructions is often the most-consuming process, which is involved in analysing the handler that used to interpret every bytecode instruction.

Numerous approaches have been proposed to protect VM handlers from reverse engineering. Most of them aim to increase the diversity of program behavior by obfuscating the handler implementation [**?**] or iteratively transforming a single program multiple times using different interpretation techniques [**?**; **?**]. However, all prior work employ a fixed strategy where each bytecode is deterministically translated to a fixed set of native code. Such techniques are vulnerable for programs protected under the same obfuscation technique. In particular, an attacker can reuse the knowledge (termed ***analysis knowledge*** in this paper) of the handler implementation obtained from one program to attack another program.

This paper resents DCVP (Code Virtualization Protection with Diversity), a enhanced VM-based code obfuscation system to address the issue of reusing analysis knowledge. We employ a technique called Instruction Set Randomization (ISR) [**?**] to randomly change the *opcodes* of bytecode instructions and their semantics, so that the mapping between bytecodes and their handlers will vary across programs. The randomization itself, however, is not sufficient for providing stronger protection, because it is easy to be bypassed due to the non-uniform distribution of bytecode instructions (e.g. the more frequent a bytecode is used, the more likely the relation between the bytecode and its handlers can be obtained from other programs). To overcome this issue, DCVP partitions the protected code region into several parts where the mappings of bytecode instructions and their handlers in each part are different. As a result, the same bytecode instruction in different parts of the bytecode program is likely to have different semantics. It is to note that this paper focuses on protecting code against reusing analysis knowledge for code reverse engineering. Like any code obfuscation technique, malware developers could also exploit our technique to protect malicious programs, but preventing this is outside the scope of this work.

The key contribution of this paper is a countermeasure to address the issue of reusing *analysis knowledge* to perform code reverse engineering across programs. We compare our approach against VMProtect [**?**] and Themida [**?**] on `md5.exe`, `gzip.exe`, `bcrypt.exe` and `mat_mul.exe` applications. Experimental results show that DCVP is able to provide stronger protection at the cost of little extra overhead. DCVP is similar to VMProtect in temporal and spatial overhead, and they are all better than Themida. And on the basis of execution time costs caused by code virtualization, the growth of DCVP partitions' number hardly affects the time overhead.

The rest of the paper is organized as follows. In Section 2, we illustrate the internals of VM-based obfuscation. Section 3 gives our threat model, which describes the analyst's capabilities and goals, and we illustrate the process of reverse engineering a VM-obfuscated program. We motivate our work in Section 4. In Section 5, 6, 7 and 8, we present the details of DCVP. Section 9 evaluates DCVP for its effectiveness and its spatial and temporal overhead. Section 10 presents the related work, and finally presents the conclusion and discusses the future work.

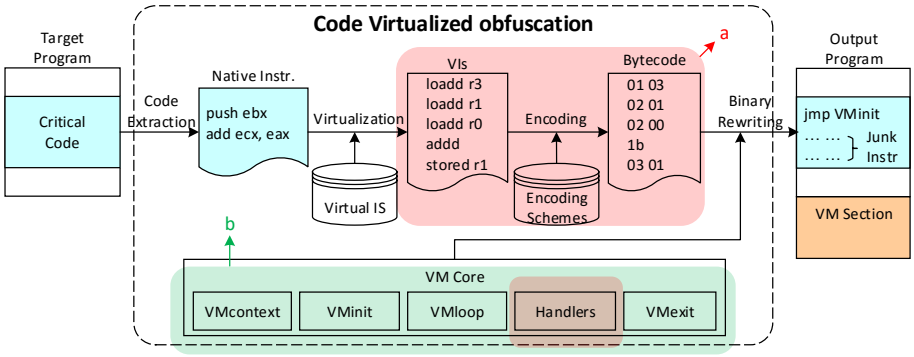

Fig. 1. A representative architecture for VM-based obfuscation. The main work of this paper is to improve the core steps of VM-based protection (areas marked as "a" and "b"). In the first region (a), we partition the protected code region into different segments, and obfuscate the `bytecode handlers` to generate multiple implementations for each `handler`. In the second region (b), we use a number of obfuscation and anti-taint analysis technologies to protect the important components of the VM core.

2. BACKGROUND

Visualization techniques is widely used to protect software programs from unauthorized analyses. Examples of VM-based code obfuscation tools include VMProtect [**?**], Code Virtualizer [**?**] and [**?**]. Code obfuscation often comes at a cost, with bloating code size and longer execution time. To minimize the overhead, in practice only critical parts of the software will be obfuscated [**?**]. VM-based protection works by transforming the native machine code of the protected code region into a set of bespoke virtual instructions (bytecode), which hopefully is unknown to the attacker. At runtime, the virtual instructions will be translated into native code using byte interpreters.

Figure 1 illustrates a classical VM-based obfuscation system. At the heart of this system are the virtual IS (Instruction Set) and the set of interpreters used to translate the IS to native code. Interpretation of virtual instructions follows the classical *decode-dispatch* approach [**?**], using a bundle of `handlers` and a `VMloop`. Here, the `VMloop` is the execution engine which fetches and decodes a bytecode instruction and then dispatches a `handler` to interpret instruction. `VMcontext`, which contains hardware-independent virtual registers and flags. At runtime, the virtual registers and flags will be mapped down to the underlying hardware, and the `VMInit` is responsible for saving the native context and initializing the `VMcontext`. In comparison, `VMExit` restores the native context when exiting VM. Finally, these VM components will be assembled into a new section and attached to the end of the target program through binary rewriting. This work targets two key components of the VM-based obfuscation architecture, highlighted as two regions Figure 1 (a and b). Our approach divides the protected code region to different sections. It generates multiple implementations for each bytecode handler using code obfuscation techniques. Different implementations of the same bytecode handler will produce an identical output for a given virtual instruction, by they follow different execution paths and exhibit diverse behavior during runtime. We further enhance the strength of the protection by using a number of obfuscation and anti-taint analysis technologies to protect the important components of the VMCore.

3. THREAT MODEL

In our threat model, we assume an attacker owns a copy of the target application and can run it in a malicious host environment [**?**], aka the white-box attack context [**?**; **?**]. In such an environment, the adversary has full privileged accesses to the system.

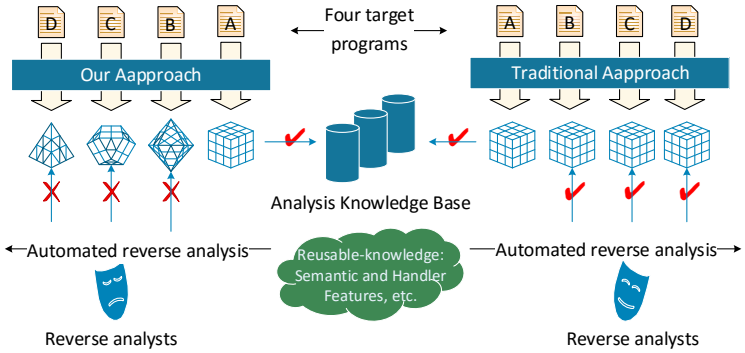

Fig. 2. The process of reusing attacking knowledge for code reverse engineering. Here we have four different target programs, A, B, C and D. In the right side of the scenario, all programs are obfuscated with a code obfuscation scheme that a virtual instruction will be deterministically translated to a fixed set of native code. This allows an attacker to reuse knowledge obtained from one program to efficiently reverse engineer other programs. In another scenario, the mapping between virtual instructions and native code is different for different programs. In this way, the attacker is unable to reuse the previously extracted knowledge to perform reverse analysis across programs.

We also assume the adversary can use static and dynamic analysis tools (such as, `IDA` [**?**], `OllyDbg` [**?**] and `Sysinternals Suite` [**?**]) to trace and analyze instructions, monitor registers and process memory, and modify instruction bytes and control flows at runtime, etc. Prior work has demonstrated that these are reason assumptions [**?**]. At present, there are two preliminarily used methods to attack VM-based protection systems, which are presented as follows.

The first is attack based on the virtual execution analysis proposed by Rolles et al. [**?**], which requires an analyst to have a certain understanding of the principle of code virtualization. By dynamically tracking the execution process of virtual interpreter to extract key bytecodes and handlers, and then through the analysis and simplify eventually recovering the original program's logic. Nicolas Falliere [**?**] presented an example of the above analysis process which is used to analyze the Trojan.Clampi protected by VMProtect [**?**]. This type of attack method is closely related to the principle and structure of the code virtualization, and has the most realistic and comprehensive results.

The other one is attack based on the behavior and semantic analysis, this type of attack method can be used to attack not only code virtualization protection but also other confusion methods. Coogan et al. [**?**] puts forward a behavior based analysis method, which aims to analyze the important behavior of code, but it does not pay attention to how to restore the original code. Yadegari et al. [**?**] propose a method based on semantic analysis, which use taint propagation to track the flow of inputs values, and semantics-preserving code transformations to simplify the logic of the instructions. This type of method has wider applicability, but it is hard to get a comprehensive analysis results.

4. MOTIVATION

Figure 2 depicts an reverse analysis scenario where an analyst can reuse the *analysis knowledge* to attack applications protected by the same VM-based code obfuscation scheme. In this example, there are four different programs to be protected, labelled as A, B, C and D. In the right side of the diagram, all the four programs are protected using an identical set of virtual instructions and bytecode handlers. Under this setting, an experienced analyst would be able to use the knowledge of the mapping of virtu-

al instructions and bytecode handlers obtained from one program to reverse-engineer the other three programs. Bear in mind that, uncovering the mapping between virtual instructions and native code is often the most time-consuming process for attacking VM-based code obfuscation. Having able to reuse the attacking knowledge thus can significantly reduce the cost involved in the attack. In another scenario, the translations between virtual instructions and native code vary among programs. Therefore, the knowledge obtained from one program will be in inapplicable to others. This forces the analyst to start from the scratch when reverse engineering a new program. This example shows that shuffle the relationship between the virtual instructions and bytecode handlers can significantly increase the effort and cost involved in performing the attack. In the remainder section, we describe how we can construct such as scheme in details.

5. OVERVIEW

DCVP consists of four components described as follows.

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 and 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 is discussed in Section 6.

Native code translation. We develop a tool to automatically translate the native machine code into virtual instructions and stored as bytecode. This is detailed in Section 7.

Bytecode diversification. The generated virtual instructions will be diversified using a special encoding scheme. Each protected code region is partitioned to multiple segments and the opcodes of the virtual instructions in each segment will be mapped to different native code. This means that a mapping from opcode to native code found at one segment is likely to be inapplicable for 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 adopt anti-taint analysis to protect the core component of the VM. This is the key component of DCVP which is presented at Section 8.

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

6. VIRTUAL INSTRUCTION SET AND HANDLERS

As the basis of a VM-based obfuscation system, virtual IS and the `handlers` should be the first considerations when designing such an system. The challenge lies in devising a feature-complete virtual IS that is Turing-equivalent to native IS, which means that any native instructions could be substituted with the virtual instructions. Virtual instructions ultimately will be executed by the hand-crafted `handlers`; 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. In this paper, we choose stack-based architecture for the VM-based obfuscation system for the following reasons:

— In a stack-based VM, operations are carried out with the help of stack, where operands and results of operations are stored. This simplifies the addressing of operands and ultimately simplifies the implementation of `handlers`.
— The process of converting native x86 instructions to virtual instructions is simpler.
— Stack-based VMs require more virtual instructions for a given computation; this makes the instructions more complex and conforms to our objective of impeding 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 are 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 for data transfers. `load` instructions are for pushing operands into stack, and `store` instructions are for popping results out of the stack and store the results into the virtual context.
— Arithmetical and logical instructions. The variants of these kind of instructions are much smaller than their native ones, as the addressing mode of operands is simpler and uniform, i.e., stack-based addressing.
— Branch instructions for changing the control flow of bytecode program.

Other instructions that are 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.

6.1. "`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 [**?**] 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 I shows the virtual instructions of `load` and `store` and their corresponding `handlers`. In table I, there exists 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

Table I. 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.

Table II. 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`

Note: Since the operations of these instructions could be 8-bit, 16-bit, or 32-bit, we design a virtual instruction for each of the operations of different operand size.

(memory address is stored in a register or presented as an expression). An example in table V illustrates the situation of using of `load_ms`.

6.2. Arithmetical and Logical Instructions

Arithmetical and logical virtual instructions are used to execute the aimed operations and they need not 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` operation as an example to illustrate. Table II 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 a virtual instruction for each of the operations of different operand size.

6.3. 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 DCVP, 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.

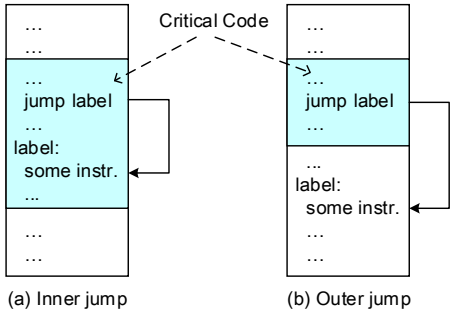

Fig. 3. The destination of a branch instruction could be in the critical code or outside the critical code. Branch instructions include `jmp`, `jcc`, `call`, and `retn`.

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 III classifies different forms of `jmp/jcc/call/retn` instructions considering the above two categories.

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

Table III. 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.

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 IV presents the above two virtual instructions of *direct* `jmp` and their handlers.

Table IV. 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.

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 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.

6.4. 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 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.

Table V. 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`.

7. NATIVE INSTRUCTIONS TO VIRTUAL INSTRUCTIONS

At obfuscation time, we first convert native instructions into virtual instructions. This process follows a three-phase fashion: first, loading the operands into stack with `load` virtual instructions; then, executing the aimed operation; and finally, storing the result into virtual context or a certain memory address with `store` virtual instructions. Table V gives some examples of native instructions and their corresponding virtual instructions.

Data transfer instructions are mainly mapped into `load` and `store` instructions. Typical examples of these instructions are `mov`, `push`, and `pop`. Arithmetical and logical instructions strictly follow the three-phase processing. Branch instructions are mapped into a `load` instruction followed by a branch virtual instruction. Native instructions with complex addressing modes are processed iteratively with the above virtual instructions, for example, the `"mov eax, dword[esi+4]"` instruction in Table V.

8. 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 virtual interpreter of 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 a virtual instruction. In practice, we assign each virtual instruction a distinct ID as its *opcode*. The ID is used by `VMloop` as an index to find the address of the `handler` of the virtual instruction in the address table recording the addresses of each `handler`. Since the number of virtual instructions is less than 256, thus one byte is sufficient to encode their IDs. As for the *operands*, since they could be of different size[1], we use one, two, or four bytes to encode them correspondingly. Figure 4 shows some examples of virtual instructions and their bytecode. The figure also demonstrates how `VMloop` fetches and interprets the bytecode instructions.

[1]The *operand* of a virtual instruction could be an index for virtual register, an immediate value, or a memory address. They could be of different size: a virtual register index being 8 bits, an immediate value being 8/16/32 bits, and a memory address being 32 bits.

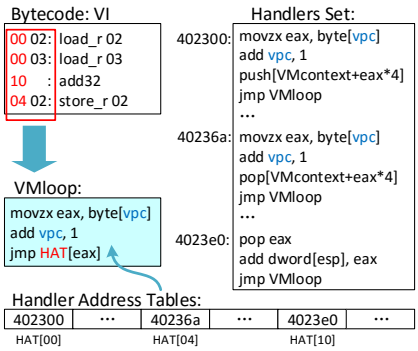

Fig. 4. Examples of some virtual instructions and their bytecode. Each virtual instruction is encoded into a bytecode instruction, which consists of an *opcode* and optionally an *operand*. The bytecode instructions feed into `VMloop` and the *opcode* of each bytecode instruction is used by `VMloop` as an index to find the address of the corresponding handler in the HAT (Handler Address Table).

8.1. Randomize the Semantics of Bytecode Instructions

From the above demonstration, if an analyst gets known the semantics of a bytecode instruction, the next time she encounters it, she does not bother to analyze its `handler` once again to figure out what it does[2]. For example, in Figure 4, the bytecode instruction `"10"` means an addition operation through analyzing `Handler_4023e0`. The next time we encounter a bytecode instruction of `"10"`, we could say that it does an addition operation immediately.

To mitigate the effect of reuse of previously obtained *analysis knowledge*, we randomize the semantics of virtual instructions. According to the encoding scheme we adopt, it is easy to achieve this goal. The idea is to change the relationship between the IDs (*opcodes*) and the virtual instructions, which is similar to [**?**]. Every time to encode the virtual instructions, the IDs are first shuffled once. Then the shuffled IDs are used to encode the virtual instructions. The addresses of `handlers` are also filled into the handler address table accordingly.

8.2. Partition Bytecode Program

With the randomization of the semantics of bytecode instructions, an analyst can not directly reuse her *analysis knowlege* to work out what a bytecode instruction actually does. However, the effect of the randomization could be easily bypassed. As shown in table V, the frequencies of virtual instructions are not uniform, where `load_r` and `store_r` are two of the most frequently used virtual instructions. Thus, an analyst could infer the semantics of bytecode instructions based on the non-uniform frequencies of *opcodes*.

8.2.1. Bytecode Program Partition Design. To frustrate the inferences based on the frequencies of *opcodes*, we partition all the generated virtual instructions into several parts, each part been encoded differently. Specifically, during obfuscation, instead of encoding the generated virtual instructions all at a time, we encode those resulted parts separately. And prior to each encoding process, we first randomly shuffle the IDs of virtual instructions and then use the results for encoding. The effect of the shuffles is that an identical *opcode* in different parts of the bytecode program probably reveals different semantics, thus the frequencies of *opcodes* are obscured . Figure 5 shows an

[2]Since `handlers` could be mutated to hinder analysis, it saves an analyst a lot of time end effort without bothering to analyze them once again.

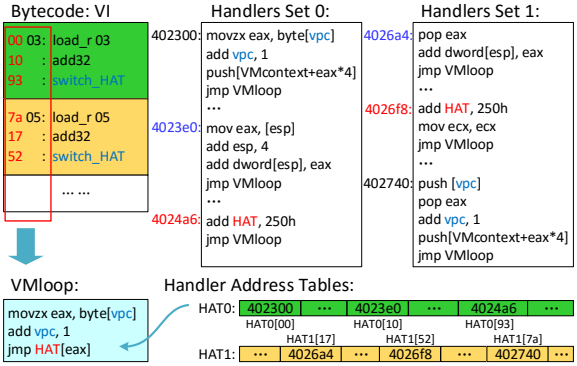

Fig. 5. Example of partitioning virtual instructions into several parts (two parts in this figure). Virtual instructions in different partitions are encoded differently and interpreted using different `handlers` set. The number of HAT increases accordingly. To switch the currently used (by `VMloop`) HAT to the next one, we add a new virtual instruction `switch_HAT`. The *operand* of `switch_HAT` is the size of a HAT.

example of partitioning the virtual instructions into two parts. The opcode of a virtual instruction is probably encoded differently in different parts. For example, `load_r` is encoded into `"00"` in the first part, while `"7a"` in the other.

ALGORITHM 1: Partition Bytecode Program

Input: VIs of critical code segment, partition number N.
Output: 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 − − ;
end while
```

Algorithm 1 demonstrated the implementation of the partition encoding. Partition the bytecode program, we will randomly divide virtual instructions into N partitions and obfuscate the original HAS (Handler Set) for N times to get N sets of `handlers`, and the detailed HAS obfusction approach are demonstrated in section 8.3. 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 can be interpreted by system's virtual interpreter.

As the *opcodes* of bytecode instructions are used by `VMloop` as the indexes for the addresses of their corresponding `handlers`, and different partitions are encoded differently, each partition needs their own HAT. At the end of a partition, the HAT used by `VMloop` should be switched to the HAT of the next partition. This is done by a new

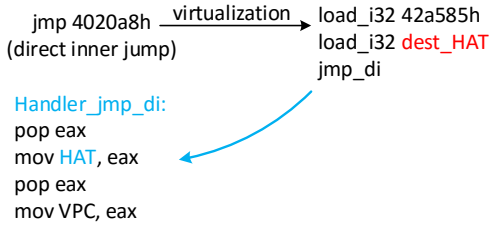

Fig. 6. The virtualization of a *direct inner* transfer instruction with HAT switching. The address of the destination HAT is pushed into stack by `load_i`, and is assigned to the HAT pointer used by `VMloop` at runtime.

virtual instruction - `switch_HAT`. Since `switch_HAT` is always added to the end of a partition and the orders of HATs are in accordance with that of the partitions, `switch_HAT` needs to add the size of a HAT to the HAT pointer used by `VMloop` (as `Handler_4024a6` does in figure 5). In our prototype, the number of `Handlers` is `148` and the address of a `Handler` is `4` bytes, thus the size of a HAT is `592` (`250h in hexadecimal`) bytes.

The switchings of HATs is not limited to the end of partitions. A branch instruction also causes the switching when its destination resides in a different partition. Branch instructions change the control flow of a program through changing the VPC. When encounter such an instruction, we cannot simply append a `switch_HAT` to it, since the `switch_HAT` may not get interpreted by `VMloop` if the VPC is changed to a location in another partition. Hence, we put the code for switching inside the `Handlers` of the branch instructions. Here, the branch instructions indicate the *direct inner* ones, as *direct outer* branches and *indirect* branches all leave the virtual context and need not to worry about the switching of HATs. During protection, for each *direct inner* branch instruction, we first calculate its destination, and then figure out which partition the destination resides. The address of the HAT of that partition is pushed into 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 a *direct inner* branch instruction, compared to that in table IV.

8.2.2. Security Analysis of Partition Design. Assume there is an attacker now, which uses the attack method based on virtual execution that introduced in section 3 to reverse the partition bytecode 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 a special encoding schemes and its bytecode program is divided into multiple partitions. So the attacker must first obtain the partition informations of critical code to further reverse analysis, but it is difficult for him. After the NIs is 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 need to analyze the bytecode of each partition and extract the semantic

information implied by the `handlers`. Because the bytecode of each partition has a 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 DCVP is more like this. Unique partition configuration and different `handler` sets, so that the attacker cannot achieve the batch automatic attack by building attack knowledge base. The attacker has to spend a lot of time to analyze every detail for each program.

8.3. 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 looks different. An analyst cannot immediately recognize them as the the same one and needs to analyze each of them, which increases the workload of the analyst. At the same time also can prevent the attacker use attack knowledge base to match these `handlers` to achieve automated reverse analysis.

ALGORITHM 2: Virtual Interpreter Obfuscation

Input: The original HAS, partition number N.
Output: 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 − − ;
end while
```

The obfuscation approach we take is shown in Algorithm 2. The number of HAS obfuscated is determined by the number of partitions. Our system will randomly select several methods from the obfuscation method library which contains the junk instructions injection, equivalent instruction substitution [**?**], code out-of-order [**?**] and control flow flattening [**?**]. Then the system will use the selected method in a random order to obfuscate the `handler`. Finally we have multiple equivalent but different forms of HASs. We will also adopt some anti-taint analysis techniques (some details are presented in section 8.4) to protect the HASs that after obfuscated. This can effectively prevent the virtual interpreter from being attacked by some de-obfuscation methods.

For example, HAS (Handler Set) as an original `handler` set that consists of m `handler`. We use a HAT to store the address of these `handlers`, and their index corresponds to the *opcode* of virtual instruction. HAS will be obfuscated for n times with different strategies, n is dependent on the number of partitions. Then we get multiple HASs and which are semantic equivalence but have different forms. At this time, all of the equivalent `handlers` still have the same index. This is a type of insecure and direct mapping relationship. Therefore, according to the method that partition bytecode program and randomized the semantics of bytecode instructions in the upper section 8.2, we first randomly shuffle the IDs of virtual instructions and then use these results to generate a new HAT for each partition (as shown in figure 5). The effect of shuffles is

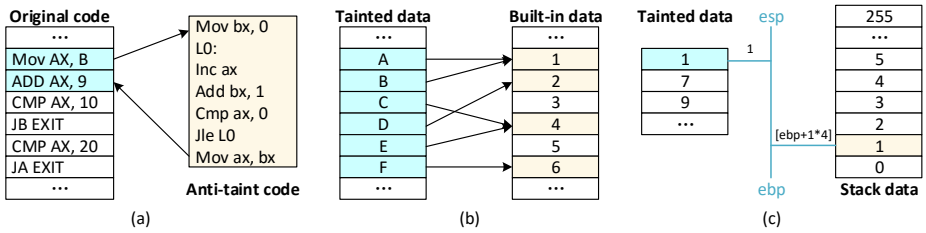

Fig. 7. Examples of three anti-taint analysis techniques. Respectively as, (a) the transfer of naive model, (b) the transfer of control dependencies and (c) the indirect transfer of stack pointer.

that an identical *opcode* in different parts of the bytecode program probably reveals different semantics. The relationship of these equivalent `handlers` in different HASs should be:

$$HAS_1(i) \Leftrightarrow HAS_2(j) \Leftrightarrow \ldots \Leftrightarrow HAS_n(k),\ 1 \leq i, j, k \leq m.$$

This various semantics of bytecode instructions and different forms of `handlers` can effectively prevent the attacker from using the attack knowledge base matching to realize the automated reverse analysis. The attacker has to spend a lot of time to analyze every detail.

8.4. 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, Coogan et al. [**?**], who use equational reasoning about assembly-level instruction semantics to simplify away obfuscation code from execution traces of emulation-obfuscated programs. Yadegari et al. [**?**], which use taint propagation to track the flow of values from the programs inputs to its outputs, and semantics-preserving code transformations to simplify the logic of the instructions that operate on and transform values through this flow.

However, the work of Coogan et al. is based on equational reasoning about assembly level instruction semantics, and it is difficult to simplify the complex equation, that making it hard to separate out the different components of nested loops or complex control flow. The work of Yadegari et al. can be effective even when applied to previously unseen obfuscations, but code simplification should be first to identify input-to-output data flows. This rely on the taint propagation and analysis to identify the explicit flow of values from inputs to outputs, then use control dependence analysis to identify implicit flows. Therefore, these methods may be impeded by enforcing dataflow obfuscation to the handling procedures [**?**]. We adopt some anti-taint analysis techniques to protect the data flow of the `handlers` from taint analysis. Specific ways are as follows:

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.

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 non 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.

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

The above three methods can prevent the spread of tainted data by laundering tainted data and resist the taint analysis effectively .

9. EVALUATION

In this section, we will evaluate the effectiveness of our method for mitigating the effect of reuse of *analysis knowlege* and the spatial and temporal overhead of our method.

9.1. Effectiveness Evaluation

DCVP is effective for impeding reverse analysis by invalidating existing *analysis knowledge*. From Section 2, we learned that understanding the semantics of bytecode instructions is essential for reverse engineering a VM-obfuscated program. Semantics are encapsulated in `handlers`, and the work of extracting them form `handlers` is tedious and error-prone. Therefore, it will save analysts lots of time and energy if the semantics of bytecode instructions are accessible, without bothering to trace and analyze the `handlers` once again. DCVP's aim is to frustrate this attempt and force analysts to analyze the `handlers` every time. Through randomizing the semantics of bytecode instructions, the same bytecode instruction probably means different obfuscated instances, and even different in the same obfuscated program by adopting different encoding schemes for different partitions of the bytecode program, which can largely confuse analysts and increase their workload.

Assuming that the number of virtual instructions, or the number of `handlers` in other words, is H, then the probability of a bytecode instructions in two obfuscated programs having the same semantics is $\frac{1}{H}$. The total number of distinct shuffle of the *opcodes* is $H!$. In an obfuscated program, supposing the bytecode program is partitioned into N parts, then the probability that a bytecode instruction in different partitions is mapped to the same native code is $\left(\frac{1}{H}\right)^{N-1}$. Therefore, we believe DCVP can effectively invalid the *analysis knowledge* about the semantics of bytecode instructions.

9.2. Overhead Evaluation

To evaluate the spatial and temporal overhead of our method, we implemented a prototype, namely DCVP, for obfuscating x86 PE executables on the Windows platform. In the implementation, we devised `148` virtual instructions and their corresponding `handlers`. We conducted all the experiments on a Dell Optiplex 960h with an Intel®Core™ 2 Duo Processor E8400 at 3.00GHz with 4.00GB of RAM. The operating system environment is Windows 7 Enterprise.

We use DCVP to protect four x86 PE executables, namely `md5.exe`[3], `gzip.exe`[4], `bcrypt.exe`[5], `mat_mul.exe`[6]. The first three are used to process a text file (`test.txt`) of 10KB and `mat_mul` is used to calculate the product of two 5×5 matrices. Table VI

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

Table VI. 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: The 5th and 6th column gives the number of instructions in the entire program and these critical functions respectively, The 7th column gives the proportion of critical code. We use Pin [**?**] to count the number of the dynamically executed instructions in the critical functions and the results are shown in the last column. We chose only 60 instructions of `mat_mul` is special to verify the impact of DCVP on the program overhead when protecting a small amount of code. The results are shown in figure 9 and the impact is not obvious.

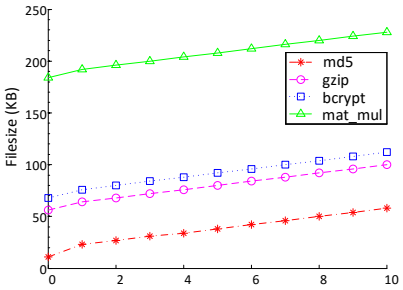

Fig. 8. The impact on file size (KB) of DCVP. The file size slightly increased with the increase of number of partitions

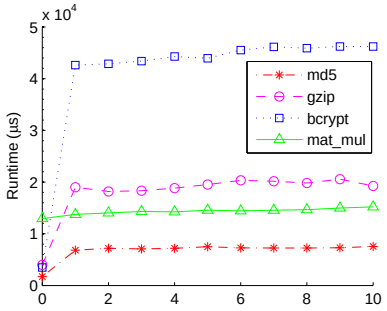

Fig. 9. The impact on runtime performance (μs) of DCVP with different partitions.

shows the statistics of these executables. For each program, we choose a piece of critical code to protect, as shown in table VI. The programs are protected 10 times, each time with a parameter that specifies a different number (1∼10) of partitions.

Figure 8 shows the size of the obfuscated programs. The horizontal axis specifies the number of partitions in the obfuscated programs and "0" means the original program. As the partitions increase, the increased bytes mostly come from the added HATs and HAS. Since the size of a HAT is only 592 bytes, the sizes of the HATs increase slowly. Besides, the sections in PE executables are aligned to a value (4096 or 512 usually)[**?**], the filesize of the program is mainly affected by the number of HAS, and it increases with the increase of partition's number regularity.

To evaluate the runtime overhead that DCVP introduces, we run the obfuscated programs for several times and calculate the average execution time of them: `md5`, `gzip`, and `bcrypt` are used to process a text file (`test.txt`) of 10KB; `matrix_mul` is used to calculate the product of two 5×5 matrices. The average execution time is shown in figure 9. Among them, `bcrypt` has the largest increase of execution time from original program[7] to the obfuscated program with one partition. This resulted from that the critical instructions in `bcrypt` is executed much more times than others (as shown in the last column in table VI). Besides, the execution time changes little as the number of partitions increases. From section 8.2 we can learn that if the number of partitions increases by one, the program only needs to execute an extra `handler` to interpret the `switch_HAT` instruction. The introduced runtime overhead is negligible. In some

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

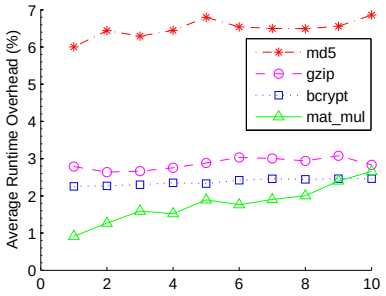

Fig. 10. The average runtime overhead per dynamically executed critical instruction.

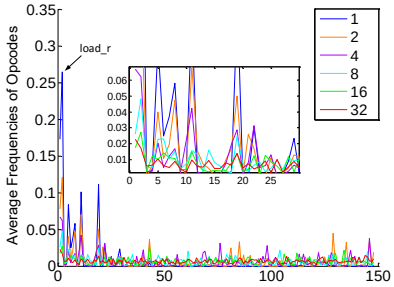

Fig. 11. The average frequencies of *opcodes*. The horizontal axis specifies the *opcodes*.

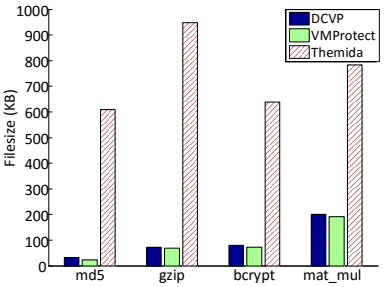

Fig. 12. The comparison of impact on file size (KB) with VMProtect and Themida.

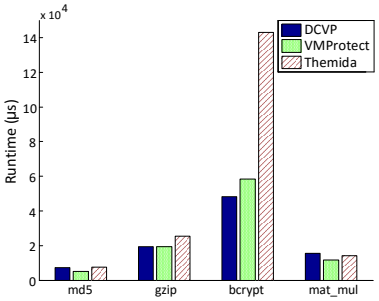

Fig. 13. The comparison of runtime performance (μs) with VMProtect and Themida.

situations, the execution time of an obfuscated program with more partitions may have a slightly lower runtime overhead. This probably results from the locality of reference[8].

We evaluate the average runtime overhead per dynamically executed instruction. We use T_{ob} to denote the execution time of a obfuscated program, T_o for that of the original program, and C_e for the count of the critical instructions been dynamically executed. The average runtime overhead per dynamically executed instruction is calculated by

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

The results are shown in figure 10. From this figure, we can learn that `md5` has the largest average runtime overhead per dynamically executed instruction. The reason is that the critical code of `md5` is full of arithmetical and logical instructions, which takes a longer time to interpret.

We also put the four target programs together and count the average frequencies of *opcodes*. We take the obfuscated programs with 1, 2, 4, 8, 16, and 32 partitions for comparison. The results are presented in figure 11. As we can see, as the number of partitions increases, the frequencies of *opcodes* tend to be closer. When the partition number is 1, we can easily get the instruction that has the highest frequency is "`load_r`", since it is the most commonly used instruction.

Finally, we use two commercial code virtualization protection system for comparison, VMProtect [**?**] and Themida [**?**]. Figure 12 shows the impact of the three virtualization protection system on the file size of the four target programs. The cost of Themida is

[8]Locality of reference. https://en.wikipedia.org/wiki/Locality_of_reference

much greater than the other two system, and the impact of DCVP and VMProtect is similar. This result should be related to the design of virtual instructions and `handlers`. Runtime overhead as shown in Figure 13. In general, the effects of the three protection systems are similar. Special, the runtime overhead of `bcrypt` that protected by Themida is far greater than the other target programs. This may be related to the design of the Themida and the executed number of critical instructions in `bcrypt`. According to above comparison, we can see that DCVP is similar to VMProtect in temporal and spatial overhead, and they are all better than Themida.

10. RELATED WORK

Software protection is used to protect the intellectual property encapsulated within software programs from been understood and modified, by transforming target program into a more obscure and hard-understanding one. The earliest published works in software protection can be traced back to 1980, Kent [**?**] addressed the security requirements of software vendors: protection from software copying and modification. During the past decades, numerous approaches of software protection have been proposed, e.g. junk instructions, equivalent instructions [**?**], packer (such as, `ASProtect` [**?**], `UPX` [**?**]), code encryption, control flow and data flow obfuscation [**?**; **?**; **?**], etc. These conventional approaches alone provide only limited obscurity, and one protection system usually integrates several of these and makes them work together to provide a certain level of obscurity.

In recent years, there have been many different code protection technology. Such as, control flow integrity [**?**; **?**] provide a strong protection against modern control flow hijacking attacks, including those based on Return-Oriented Programming (ROP). Code Randomization, Stephen Crane et al. [**?**] presents a practical, fine-grained code randomization defense, called Readactor, resilient to both static and dynamic ROP attacks. At the same time, the code reverse analysis technology is developing. Symbolic and concolic execution [**?**] and Data flow tracking and tanit analysis [**?**; **?**; **?**] can find important applications in a number of security-related program analyses, including analysis of malicious code. This paper focuses on the code virtualization protection and its reverse analysis technology.

Code virtualized obfuscation, has recently been used to protect software from malicious reverse engineering [**?**; **?**; **?**; **?**; **?**]. In Section 2 and 3 we have already introduced the details of VM-based obfuscation and possible attacks. Existing researches on VM-based obfuscation focus on foiling reverse analysis. TDVMP [**?**] obfuscates `handlers` to generate multiple semantics-equivalent but appearance-different ones and embeds them into target program. At runtime, it will execute different instructions since the execution of each `handler` is chose from the several candidates randomly, i.e., the obfuscated program has a per-process execution diversity. This is especially useful for defeating dynamic analysis. Fang et al. [**?**] presented a multi-stage obfuscation method that iteratively transforms a program for many times in using different interpretations. DCVP focuses on invalidating the analysis knowledge about the semantics of bytecode instructions and preventing from quick and large scale reverse analyses, which is orthogonal to the above approaches and is complementary to them.

Has been also have some deobfuscation techniques of code virtualization was put forward. Representative like, Sharif et al. [**?**] used dynamic data-flow and taint analysis to identify data buffers containing the bytecode program and extract the syntactic and semantic information about the bytecode instructions. Coogan et al. [**?**] proposed an approach to identify instructions that related to system calls, and automatically extract an approximate dynamic trace of the original code. Yadegari et al. [**?**], proposed an approach to track the flow of inputs values, and then use semantics-preserving code transformations to simplify the logic of the instructions. These approach, however, or

can only extract some execution characteristics of target program and can not restore the structure of the original code completely, or needed taint analysis to track and analyze the data flow, and this process may be impeded by enforcing dataflow obfuscation to the handling procedures [**?**]. These methods are limited more or less.

DCVP adopts ISR (Instruction Set Randomization) techniques while generating randomized and distinct virtual instruction sets. ISR has been used to prevent code injection attacks by randomizing the underlying system instructions [**?**; **?**; **?**]. In this approach, instructions are encrypted with a set of random keys and then decrypted before being fetched and executed by the CPU. ISR is effective for defeating code injection attacks but cannot prevent from reverse engineering attacks. As in our attack model, software programs are executed in a malicious host environment, where attackers are able to trace and log the decrypted instructions for later analysis. DCVP employs an approach similar to ISR while generating random virtual instruction sets, by changing the relationship between the opcodes and the virtual instructions [**?**], but it never "decrypts" the virtual instructions back into their original ones. Instead, DCVP uses `handlers` to interpret the virtual instructions. and the `handlers` of virtual instructions are more complex than their corresponding native instructions. Besides, DCVP uses the multiple partitions and has the different ISRs in a single program, making the reverse analyses even more difficult and tedious.

11. CONCLUSION

In this paper, we have presented the DCVP, a VM-based code obfuscation scheme. DCVP is designed to prevent code reverse engineering attacks that use knowledge obtained from programs protected under the same code obfuscation technique. We achieve this by first partitioning the protected code region to different segments; then randomly mapping the opcode of each virtual instruction from different segments to different bytecode handlers. As a result, the mapping between a virtual instruction and native code changes from one code segment to the other. This significantly increases the diversity of the program behavior, making it much harder for an attacker to reuse knowledge obtained from other programs to attack the target application. We have evaluated our approach on a set of real-world applications and compared it against state-of-the-art VM-based code protection tools. Experimental results show that DCVP provides stronger protection (measured by the how likely a virtual instruction within two segments is mapped to the same native code or handler) at the cost of little extract overhead.

FIX:I deleted your future work sentences because I have no idea of what is talking about.

ACKNOWLEDGMENTS

This work was partial supported by projects of the National Natural Science Foundation of China (No. 61373177, No. 61572402), the Key Project of Chinese Ministry of Education (No. 211181), the International Cooperation Foundation of Shaanxi Province, China (No. 2013KW01-02, No. 2015KW-003, No. 2016KW-034), the China Postdoctoral Science Foundation (grant No. 2012M521797), the Research Project of Shaanxi Province Department of Education (No. 15JK1734), the Research Project of NWU, China (No. 14NW28), and the UK Engineering and Physical Sciences Research Council under grants EP/M01567X/1 (SANDeRs), EP/M015793/1 (DIVIDEND).