Exploiting Code Diversity to Enhance Code Virtualization Protection

Chao Xue†, Zhanyong Tang†, Guixin Ye†, Guanghui Li†, Xiaoqing Gong†, Wei Wang†, Dingyi Fang†, Zheng Wang§
†School of Information Science and Technology, Northwest University, P. R. China.
§Metalab, School of Computing and Communications, Lancaster University, U. K.

***Abstract*—Code virtualization built upon virtual machine (VM) technologies is 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, exposes a security vulnerability 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-based 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 will be less useful when targeting 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.**

***Index Terms*—Virtualized obfuscation, reverse engineering, instruction set randomization, analysis knowledge**

I. Introduction

Unauthorized code reverse engineering is a major concern for software developers. It is often exploitted by adversaries to perform various attacks, including removing copyright protection of software, taking out advertisements 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 [1], [2], [3], [4], [5], [6].

Code virtualization based on a virtual machine (VM) is emerging as a promising way for implementing code obfuscation [7], [8], [9], [10], [11], [12], [13]. This strategy 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 several steps. The attack first reverse-engineers the virtual interpreter to understand the semantics of individual bytecode instructions. Then translates the bytecode back to native machine instructions or even high-level program languages to understand the program logic [14], [15]. 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 [13] or iteratively transforming a single program multiple times using different interpretation techniques [10], [11]. 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***) of the handler implementation obtained from one program to launch the attack on another program.

We present DCVP (Code Virtualization Protection with Diversity), an enhanced VM-based code obfuscation system to address the issue of reusing analysis knowledge. We employ a technique called Instruction Set Randomization (ISR) [16] to randomly change the *opcodes* of bytecode instructions and their semantics; so that the mapping between bytecodes and their handlers varies 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 the 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 program will have different semantics.

The key contribution of this paper is a countermeasure to address the issue of reusing analysis knowledge for code reverse engineering. We compare our approach against VMProtect [8] and Themida [9] on a set of real-world Intel x86 applications and algorithms. Experimental results show that DCVP provides stronger protection at the cost of little extra overhead. It is to note that our work focuses on protecting code against code reverse engineering. Like any code obfuscation technique, malware developers could also exploit this to protect malicious

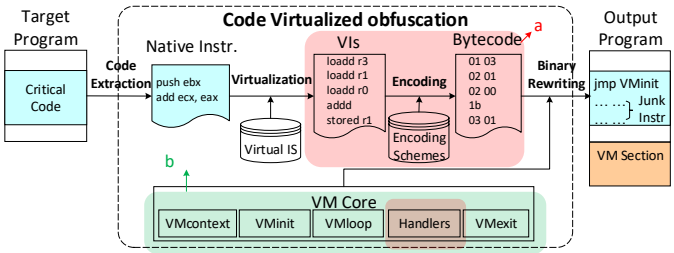

Fig. 1: A representative architecture for VM-based obfuscation. The main work of this paper is to improve the core steps of VM-based protection (areas marked as "a" and "b"). In the first region (a), we partition the protected code region into different segments, and obfuscate the `bytecode handlers` 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.

programs, but preventing this is outside the scope of this work.

II. Background

Virtualization techniques is widely used to protect software programs from unauthorized analyses. Examples of VM-based code obfuscation tools include VMProtect [8], Code Virtualizer [7] and [9]. 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 are obfuscated [17]. VM-based protection works by transforming the native machine code of the protected code region into a set of bespoke virtual instructions which are stored as bytecode in the program binary. 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 [18], 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.

Our work focuses on two key components of the VM-based obfuscation architecture, highlighted using labels 'a' and 'b' in Figure 1. 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 are semantically equivalent and will produce an identical output for a given virtual instruction; but 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.

III. The 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 [19]. Such a threat model is also known as the white-box attack [20], [21]. In such an environment, the adversary has full privileged accesses to the system. We also assume the adversary can use static and dynamic analysis tools, such as `IDA` [22], `OllyDbg` [23] and `Sysinternals Suite` [24], 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 [14], which are often available to an experienced adversary.

There are two preliminarily used methods to attack VM-based protection systems. Our work assumes an adversary can use any of or a combination of the methods to launch the attack. These two methods are described as follows.

The first technique is based on the virtual execution analysis proposed by Rolles *et al.* [15]. This requires an analyst to have a deep understanding of the code virtualization techniques employed by the obfuscation system. It works by dynamically tracking the execution process of the virtual interpreter to extract the key bytecodes and handlers, and then through the analysis and code simplification to recover the program logic. Falliere *et al.* [14] show that it is possible to perform the above analysis [8]. This type of attack method is closely related to the principle and structure of code virtualization, and has been widely adopted to analyze obfuscated malware.

The second technique is based on behavior and semantic analysis of the target program. This type of attack method can be used to attack not only code virtualization but also other obfuscation methods. Coogan *et al.* [25] propose a behavior based analysis method. Their approach aims to analyze important behavior of code, but it does not pay attention to restoring the original code. Yadegari *et al.* [26] propose a method based on semantic analysis. The method uses 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 is restricted to a small code region.

IV. 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 virtual 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

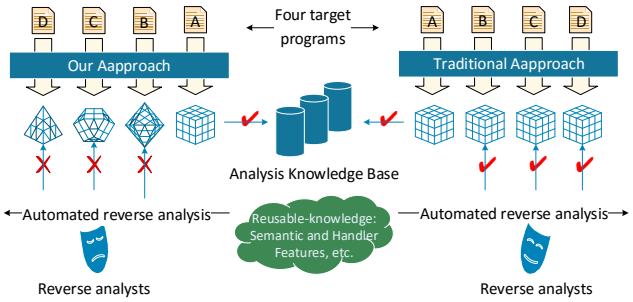

Fig. 2: The process of reusing attacking knowledge for code reverse engineering. Here we have four different target programs, A, B, C and D. In the right side of the scenario, all programs are obfuscated with a code obfuscation scheme that a virtual instruction will be deterministically translated to 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.

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 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 sections of the paper, we describe how we can construct such as scheme in details.

V. Overview

DCVP consists of four components, described as follows.

Virtual Instruction Set and Handlers. The native machine instructions within the target code region are translated into bespoke virtual instructions and stored as bytecode in the program binary. The virtual instructions will be decoded by the handlers during runtime. A virtual instruction can be decoded by multiple semantically-equivalent handlers. A brief description of the design of a virtual instruction set is given in Section VI.

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 VII-A.

Bytecode diversification. The generated bytecode 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. This is the key component of DCVP, presented at Section VII-B.

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

VI. Virtual Instruction Set and Handlers

The virtual instruction set and their handlers are the foundations of any VM-based code obfuscation system. The virtual instruction set must be Turing-equivalent to target native machine code. This means that any native instruction could be substituted with some virtual instructions without violating the semantics of the original code. Virtual instructions will be interpreted by the hand-crafted `handlers` during program execution. It is to note that the instruction `handlers` are written in native instructions.

There are two mainstream approaches for implementing a VM: a stack-based approach and a register-based one. In this paper, we choose to use the stack-based architecture to implement DCVP 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 compared to a register-based alternative.
- 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.

A näive approach to design a virtual instruction set that is semantically equivalent to the native instruction set, is to map every single native instruction to a virtual instruction. However, this would require us to implement a large number of instruction handlers as our goal is to provide multiple handlers for a single virtual instructions. We choose a different approach that has a lower implementation cost, by exploiting the characteristics of a stack-based VM. Our design choice is based on the following observation. In a stack VM, a native operation is either executed or virtualized a three-step fashion: (i) pushing the operand into the stack, (ii) executing an operation, and (iii) storing the result into the execution context. Therefore, we can use the following, smaller number of instructions to implement a virtual instruction set:

- We need `load` and `store` instructions for data transfers. `load` instructions are for pushing operands into the stack, and `store` instructions are for popping results out of the stack and storing the results back the virtual context.
- We need arithmetical and logical instructions. The number of these virtual instructions needed are smaller than their Intel x86 counterparts, as the addressing mode of operands is simpler and uniform (i.e., stack-based memory addressing).
- Branch instructions for changing the control flow of the bytecode program.

Other instructions that are not included in the above categories are defined as special virtual instructions, and are labelled as `undef`, When encountering such an instruction at time, we will first restore the native context and exits the VM. Then, we execute the undefined native instruction in the

native context and re-enter the VM to continue executing the remaining bytecode instructions.

VII. Offline Code Obfuscation

We now describe how to translate native instruction instructions to virtual instructions and store them in the bytecode format.

A. Native Instructions to Virtual Instructions

At code obfuscation time, we first convert native instructions into virtual instructions. This conversion process follows the three-step execution process in a stack-based VM, which is described in Section VI. Specifically, we first load the operands into stack with the `load` virtual instruction; then, we execute the ready to execute operation; and store the result into virtual context or a certain memory address with `store` virtual instructions.

Native data transfer instructions are mainly mapped into `load` and `store` instructions. Examples of such instructions in the x86 instruction set include `mov`, `push`, and `pop`. Arithmetical and logical instructions will be translated by strictly following the aforementioned three-phase processing, and branch instructions are mapped into a `load` instruction followed by a virtual branching instruction. Native instructions with complex addressing modes are processed iteratively using a combination of the aboved virtual instructions.

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 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.

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 3, 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.

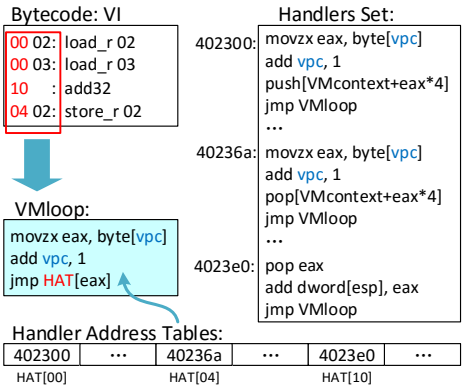

Fig. 3: Examples of some virtual instructions and their bytecode. Each virtual instruction is encoded into a bytecode instruction, which consists of an *opcode* and optionally an *operand*. The bytecode instructions feed into `VMloop` and the *opcode* of each bytecode instruction is used by `VMloop` as an index to find the address of the corresponding handler in the HAT (Handler Address Table).

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 [16]. 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.

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. 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*.

3) Bytecode Program Partitioning: To obfustrate 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 4 shows an 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.

[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.

[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.

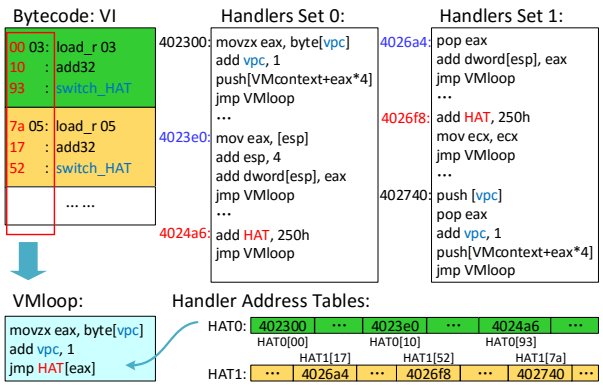

Fig. 4: Example of partitioning virtual instructions into several parts (two parts in this figure). Virtual instructions in different partitions are encoded differently and interpreted using different `handlers` set. The number of HAT increases accordingly. To switch the currently used (by `VMloop`) HAT to the next one, we add a new virtual instruction `switch_HAT`. The *operand* of `switch_HAT` is the size of a HAT.

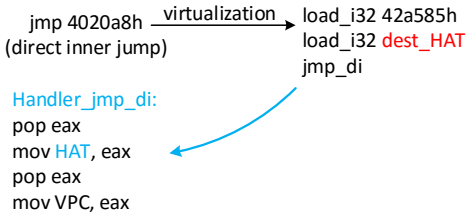

Fig. 5: The virtualization of a *direct inner* transfer instruction with HAT switching. The address of the destination HAT is pushed into stack by `load_i`, and is assigned to the HAT pointer used by `VMloop` at runtime.

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 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 4). 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 5 shows the virtualization of a *direct inner* branch instruction.

4) Security Analysis of Partition Design: 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 [27], code out-of-order [28] and control flow flattening [29]. 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 HAS. We will also adopt some anti-taint analysis techniques (some details are presented in section III) to protect the HAS 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 partitions bytecode program and randomizes the semantics of bytecode instructions described in Section VII-B2, 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 4). The effect of shuffles is 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:

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

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.

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

VIII. Evaluation

In this section, we first evaluate the effectiveness of DCVP by computing the likelihood of a bytecode instruction of two partitions to be mapped to an identical machine code. We then evaluate the overhead of DCVP in terms of code size and running time.

TABLE I: Statistics of the Target Programs.

Target Program	Version	Size(KB)	Critical Code	N1	N2	Ratio	N3
md5	2.3	11	Transform()	1327	563	42.36%	85013
gzip	1.2.4	56	deflate()	10181	153	1.50%	539082
bcrypt	1.1.2	68	Blowfish_Encrypt()	2997	54	1.80%	1735710
mat_mul	-	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 [30] 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 8 and the impact is not obvious.

A. *Effectiveness Evaluation*

DCVP is effective for impeding reverse analysis by invalidating existing *analysis knowledge*. From Section II, 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 instruction 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 having the same semantics is $\left(\frac{1}{H}\right)^{N-1}$. Therefore, we believe DCVP can effectively remove the *analysis knowlege* about the semantics of bytecode instructions.

We also count the average frequencies of *opcodes* of our benchmarks. We take the obfuscated programs with 2, 8, and 32 partitions for comparison. The results are presented in figure 6. 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.

B. *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.

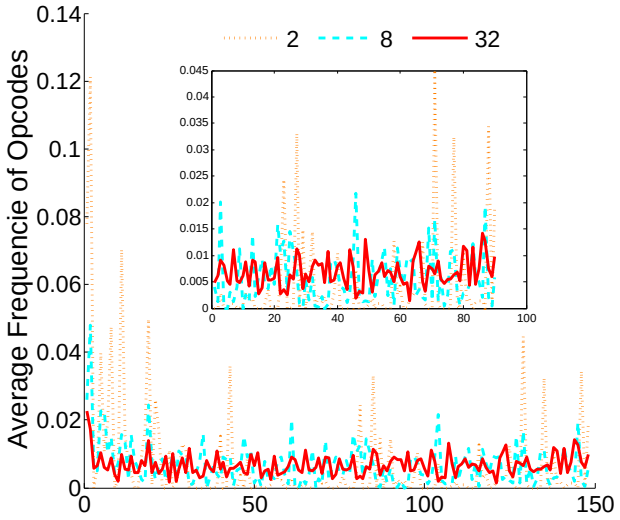

Fig. 6: The average frequency of an *opcode* is used during runtime execution. The x-axis shows the opcode ID and the y-axis shows how often (normalized between 0 and 1) an opcoded is chosen.

We use DCVP to protect four x86 PE executables, namely `md5`[3], `gzip`[4], `bcrypt`[5], `mat_mul`[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 I shows the statistics of these executables. For each program, we choose a piece of critical code to protect, as shown in table I. The programs are protected 10 times, each time with a parameter that specifies a different number (1∼10) of partitions.

Figure 7 shows the code 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)[31], the filesize of the program is mainly affected by the number of HAS, and it increases with the increase of partition's number regularity.

[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.

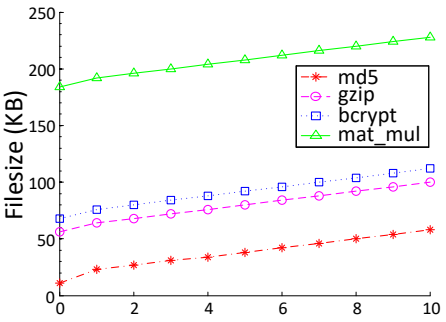

Fig. 7: The impact on code size (KB) of DCVP. The file size slightly increased with the increase of number of partitions

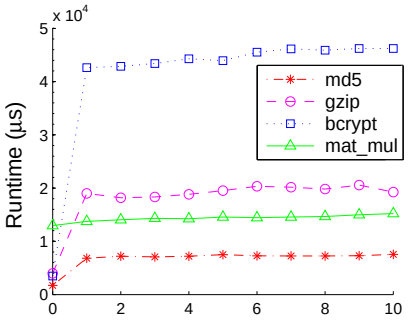

Fig. 8: The impact on runtime performance (μs) of DCVP with different partitions.

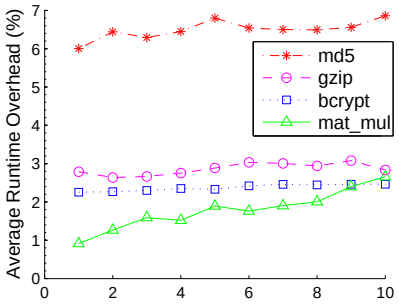

Fig. 9: The average runtime overhead per dynamically executed critical instruction.

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 8. 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 I). Besides, the execution time changes little as the number of partitions increases. From section VII-B2 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 situations, the execution time of an obfuscated program with more partitions may have a slightly lower runtime overhead. This probably is due to better data locality that leads to favourable cache behaviors.

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 9. 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.

Finally, we compare DCVP to two commercial code virtualization protection systems: VMProtect [8] and Themida [9]. Figure 10 shows the impact of the three virtualization protection system on the file size of the four target programs. The cost of Themida is much greater than the other two system, and the impact of DCVP and VMProtect is similar. This result

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

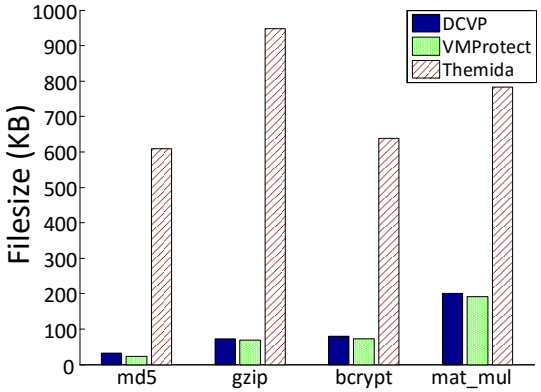

Fig. 10: The comparison of impact on file size (KB) with VMProtect and Themida.

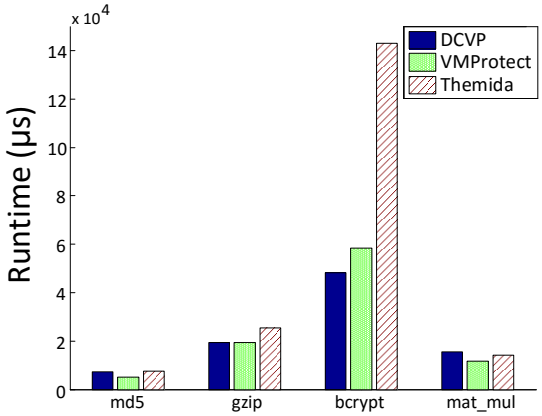

Fig. 11: The comparison of runtime performance (μs) with VMProtect and Themida.

should be related to the design of virtual instructions and `handlers`. Runtime overhead as shown in Figure 11. 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.

IX. Related Work

Code deobfuscation techniques have been proposed in recent years. The representative researches are as follows, Coogan et al. [25] 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. [26], 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 [21]. 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 [32], [16], [33]. 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 [16], 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.

X. Conclusions

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. At the core of DCVP is a novel strategy to diversify the obfuscated program behavior. 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 makes the program behavior to be less predictable, increasing the difficulty for reusing 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 at the cost of little extract overhead.

References

[1] S. Schrittwieser *et al.*, "Protecting software through obfuscation:can it keep pace with progress in code analysis?" 2016.

[2] P. Ananth *et al.*, "Optimizing obfuscation:avoiding barrington's theorem," 2014.

[3] J. Zeng and oters, "Obfuscation resilient binary code reuse through trace-oriented programming," in *CCS*, 2013.

[4] L. Luo *et al.*, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," in *FSE*, 2014.

[5] M. Preda and R. Giacobazzi, "Control code obfuscation by abstract interpretation," in *IEEE International Conference on Software Engineering and Formal Methods(SEFM)*, 2005.

[6] Z. Wu and oters, "Mimimorphism: A new approach to binary code obfuscation," in *CCS*, 2010.

[7] "Oreans Technologies. Code Virtualizer. http://www.oreans.com/codevirtualizer.php."

[8] "VMProtect Software. VMProtect. http://vmpsoft.com/."

[9] "Themida.http://www.oreans.com/themida.php."

[10] H. Fang *et al.*, "Multi-stage binary code obfuscation using improved virtual machine," in *Information Security*, 2011.

[11] M. Yang and L. S. Huang, "Software protection scheme via nested virtual machine," *Journal of Chinese Computer Systems*, 2011.

[12] H. Wang *et al.*, "NISLVMP: Improved Virtual Machine-based software protection," in *9th International Conference on Computational Intelligence and Security (CIS)*, 2013.

[13] H. Wang, D. Fang, G. Li, N. An, X. Chen, and Y. Gu, "Tdvmp:improved virtual machine-based software protection with time diversity," in *ACM Sigplan on Program Protection and Reverse Engineering Workshop*, 2014, pp. 1–9.

[14] N. Falliere *et al.*, "Inside the jaws of Trojan.Clampi," *Rapport technique, Symantec Corporation*, 2009.

[15] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, 2009.

[16] K. Kuang *et al.*, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling ," 2018.

[17] D. Geneiatakis *et al.*, "Adaptive defenses for commodity software through virtual application partitioning," in *CCS*, 2012.

[18] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, H. Zhang, and Z. Wang, "Exploit dynamic data flows to protect software against semantic attacks," 2017.

[19] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on Software Engineering*, 2002.

[20] S. Chow *et al.*, "A white-box des implementation for drm applications," in *Digital Rights Management*, 2003.

[21] Y. Jia *et al.*, "A generic attack against white box implementation of block ciphers," in *International Conference on Computer, Information and Telecommunication Systems*, 2016.

[22] "IDA Pro, https://www.hex-rays.com/index.shtml."

[23] "OllyDbg, http://www.ollydbg.de/."

[24] "Sysinternals suite, https://technet.microsoft.com/en-us/sysinternals/bb842062/."

[25] B. Yadegari *et al.*, "A generic approach to automatic deobfuscation of executable code," in *IEEE S & P*, 2015.

[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 *Trustcom/bigdatase/icess*, 2017, pp. 261–268.

[27] G. C. Nicolaou George, "Applied binary code obfuscation," Tech. Rep., 2009.

[28] M. Kayaalp *et al.*, "Efficiently securing systems from code reuse attacks," 2014.

[29] S. Blazy and A. Trieu, "Formal verification of control-flow graph flattening," vol. 27, no. 4, 2016, pp. 176–187.

[30] "Pin - A dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool."

[31] "Peering inside the PE: A tour of the Win32 portable executable file format. https://msdn.microsoft.com/en-us/magazine/ms809762.aspx."

[32] J. A. Ambrose *et al.*, "Randomized instruction injection to counter power analysis attacks," 2012.

[33] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *ACSAC*, 2010.