Exploiting Dynamic Scheduling for VM-Based Code Obfuscation

Kaiyuan Kuang[†], Zhanyong Tang[†*], Xiaoqing Gong[†], Dingyi Fang[†],
Feng Chen[†], Guixin Ye[†], Jie Zhang[†], Zheng Wang[‡]
[†]School of Information Science and Technology, Northwest University, Xi'an, 710127, P.R. China.
[‡]School of Computing and Communications, Lancaster University, UK

***Abstract*—Code virtualization building 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 scheduling structure where the program follows a single, static execution path for the same input. Such approaches, however, are vulnerable to certain scenarios where the attacker can reuse knowledge extracted from previously seen software to crack applications using similar protection schemes. This paper presents DSVMP, a novel VM-based code obfuscation approach for software protection. DSVMP brings together two techniques to provide stronger code protection than prior VM-based schemes. Firstly, it uses a dynamic instruction scheduler to randomly direct the program to execute different paths without violating the correctness across different runs. By randomly choosing the program execution paths, the application exposes diverse behavior, making it much more difficult for an attacker to reuse the knowledge collected from previous runs or similar applications to perform attacks. Secondly, it employs multiple VMs to further obfuscate the relationship between VM bytecode and their interpreters, making code analysis even harder. We have implemented DSVMP in a prototype system and evaluated it using a set of widely used applications. Experimental results show that DSVMP provides stronger protection with comparable runtime overhead and code size when compared to two commercial VM-based code obfuscation tools.**

***Index Terms*—Code virtualization; Code Obfuscation; Dynamic cumulative attack**

I. INTRODUCTION

Unauthorized code analysis and modification based on reverse engineering is a major concern for software companies. Such attacks can lead to a number of undesired outcomes, including cheating in games, unauthorized used of software, pirated pay-tv etc. Industry is looking for solutions for this issue to deter reverse engineering of software systems. By making sensitive code difficult to be traced or analyzed, code obfuscation is a potential solution for the problem.

Code virtualization building upon a virtual machine (VM) is emerging as a promising way for implementing code obfuscation [1], [2], [3], [4], [5], [6], [7]. The underlying principal of VM-based protection is to replace the program instructions with virtual bytecodes which attackers are unfamiliar with. These virtual bytecodes will then be translated into native machine code at runtime to execute on the underlying hardware platform. Using a VM-based scheme, the execution path of the obfuscated code is controlled by a virtual instruction scheduler. A typical scheduler consists of two components: a *dispatcher* that determines which bytecode is ready for execution, and a set of *bytecode handlers* that translate bytecodes into native machine code. This process replaces the original program instructions with bespoke bytecodes, allowing developers to conceal the purpose or logic of sensitive code regions.

Prior work on VM-based software protection primarily focuses on making a single set of bytecodes more complicate and uses one virtual instruction scheduler. Such approaches rely on the assumption that the scheduler and the bytecode instruction set are difficult to be analyzed in most practical runtime environments. However, research has shown that is an unreliable assumption [8] under certain scenarios (referred as *cumulative attacks* in this paper) where an adversary can easily reuse knowledge obtained from other applications protected with the same scheme to preform reverse engineering. To protect software against cumulative attacks, it is important to have a certain degree of uncertainty and diversity during program execution [9].

This paper presents DSVMP (*dynamic scheduling for VM-based code protection*), a novel VM-based code protection scheme to address cumulative attacks. Our key insight is that it will be more difficult for the attacker to analyze the implementation if the program behaves differently in different runs. DSVMP achieves this by introducing rich uncertainty and diversity into program execution. To do so, it exploits a flexible, multi-dispatched scheme for code scheduling and interpretation. Unlike prior work where a program always follows a single, fixed execution path for the same input across different runs, the DSVMP scheduler directs the program to execute a randomly selected path when executing a protected code region. As a result, the program follows different execution paths in different runs and has non-deterministic behavior. Our carefully designed scheme ensures that the program will produces a deterministic output for the same input despite the execution paths look differently from the attacker's perspective. To analyze software protected under DSVMP, the adversary is forced to use a large number of trail runs to understand how the program algorithm works. This significantly increases the cost of code reverse-engineering.

In addition to dynamic instruction scheduling, DSVMP brings together two other techniques to increase diversity of program behaviour. Firstly, DSVMP provides a rich set of bytecode handlers, which are implemented using different al-

[*]Corresponding author. Email address: zytang@nwu.edu.cn

gorithms and data structures, to translate a bytecode instruction to native code. Handlers for a particular bytecode all generate an identical output for the same input, but their execution paths and data accessing patterns are different from each other. During runtime, our VM instruction scheduler randomly selects an bytecode handler to translate a virtual instruction to the native machine code. Since the choice of handlers is randomly determined at runtime for each bytecode instruction and the implementation of different handlers are different, the dynamic program execution path is likely to be different in different runs. Secondly, DSVMP employs a multi-VM scheme so that various code regions can be protected using different bytecode instruction sets and VM implementations. This further increases diversity of the program, making it even harder for an adversary to analyze the software behavior or reuse knowledge extracted from other software products (as different products are likely to be protected by different bytecodes instructions and VM implementations).

The whole is greater than the sum of the parts. These techniques, putting together, enable DSVMP to provide stronger code protection than any of the VM-based techniques seen so far. We have evaluated DSVMP on four widely used applications: `md5`, `aescrypt`, `bcrypt` and `gzip`. Experimental results show that DSVMP provides stronger protection with comparable runtime overhead and code size when compared to two commercial VM-based code obfuscation tools: Code Virtualizer [2] and VMProtect [3].

This paper makes the following contributions:

- It presents a dynamic scheduling structure for VM-based code obfuscation to protect software against dynamic cumulative attacks.
- It is the first to apply multiple VMs to enhance diversity of code obfuscation.
- It demonstrates that the proposed scheme is effective in protecting real-world software applications.

II. BACKGROUND

VM-based code obfuscation transforms native instructions of protected code regions to virtual instructions. Virtual instructions are encoded into bytecodes which will be translated to native machine code at runtime. In the process of protection, a new VM section is inserted to the end of target program and the entry point of a protected code region is redirected to a function call to the VM. Classical VM bytecodes are based on a stack machine model where computation is performed using stack operations like push and pop. When entering the VM, context of the native program, which includes information such as local variables, function arguments, return address etc., will be stored in a VM memory space called VMContext consisting of a number of virtual registers. When exiting the VM, the native context will be restored. At the heart of the VM is an interpreter with two components: a dispatcher to determine which bytecode instruction is ready for execution and a set of bytecode handlers to translate a bytecode to naive machine code. The idea of VM-based code obfuscation is to use a set of bespoke bytecode instructions to make it harder to understand how the program works by tracking the program execution. Such protection will become invalid once the adversary figures out how bytecodes are mapped into native code.

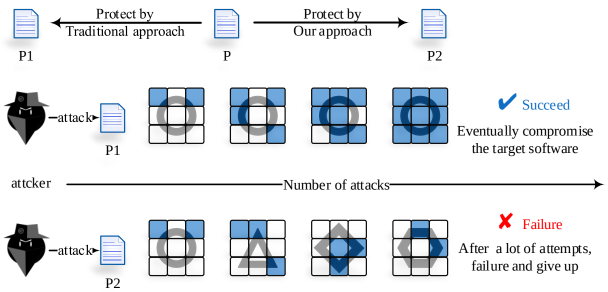


Figure 1: Diversity affects effectiveness of attack. Here a dark small square represents a piece of reusable attacking knowledge. Diverse program execution increases the difficulty of attacks.

Cumulative attacks, Figure 1 illustrates how an attacker can reuse knowledge extracted from the previous runs of the same application or other applications (that are protected using the same VM scheme) to perform attack. This is referred as *cumulative attacks* in this paper. In the first scenario, the software always follows the same execution path across multiple runs, and a few runs will allow an attacker to obtain sufficient knowledge about the program behavior. In the second scenario, the program execution path changes across different runs. As such, it will take longer and many more runs to gather enough information to perform the attack. As can be seen from this simple illustration, diversity keys to protect against dynamic cumulative attacks. This work aims to achieve this purpose.

III. THE ATTACK MODEL

In this work, we assume that the adversary holds an executable binary of the target software and can run the program in a malicious host environment [10]. We also assume the adversary has the tools and skills to access memory and registers, trace program instructions, and modify the program instructions and control flows using tools like “IDA” [11], “Ollydbg” [12] and “Sysinternals suite” [13]. The aim of the adversary is to completely reverse the internal implementation of the target program. Our goal is to increase the difficulties in terms of time and efforts for an adversary to reverse the target program implementation using VM-based code obfuscation.

A classical approach to reverse engineer a VM-protected program typically follows three steps [8], [14] described as follows. The first step is to reverse engineer the two essential components of a VM interpreter: the dispatcher and bytecode handlers. To do so, the attacker needs to locate these components and analyze how the dispatcher schedules bytecode instructions. The second step is to understand how each bytecode is mapped to machine code. The third step is to use knowledge obtained in the first two steps to recover the original logical implementation of the target program. A skilled attacker is able to use knowledge gathered from parts of the program to analyze other protected regions of the same program or other applications protected using the same VM scheme and bytecode instructions. In this work, we assume the attacker has the necessary tools and skills to implement the above attacks.

IV. Code protection scheme of Dsvmp

To address the problem of cumulative attacks, we want to introduce a certain degree of diversity and uncertainty into program execution. This is achieved through using a diversified scheduling structure (Section V) and multiple VMs (Section VI) in Dsvmp. Like other VM-based protection schemes, Dsvmp focuses on protecting critical code regions to minimize the runtime overhead. Figure 2 depicts the system architecture of Dsvmp. Code protection of Dsvmp follows several steps described as follows:

Code translation: Dsvmp takes in a compiled program binary and does not require having access to the source code. Code segments need to be protected are translated into native machine instructions (e.g. x86 instructions) using a disassembler (Step ①), which will then be mapped into a set of virtual instructions (Step ②).

Diversifying: As a departure from prior work on VM-based code obfuscation, Dsvmp employs multiple VM instruction scheduling policies where each scheduler can have more than one dispatcher and one handler can be scheduled by another handler and each virtual instructions can be interpreted by more than one handlers. A set of initial handlers will be randomly obfuscated to provide stronger protection for the particular code region (Step ③). Furthermore, each handler will be obfuscated $VMNum$ ($VMNum >= 1$) times by using the deformation engine, resulting in $VMNum$ sets of semantically equivalent handlers with different implementations and control flows (Step ④). Then, virtual instructions are encoded into $2 * VMNum$ sets of bytecodes. For each set of handlers, there will be two sets of corresponding bytecodes (details in Section V-B) (Step ⑤). Subsequently, Dsvmp constructs multiple VMs, where each VM contains one set of handlers and two sets of bytecodes (Step ⑥).

Code generation: Finally, a new section will be inserted into the program binary, which contains $VMNum$ VMs and their components such as dispatchers, VMContext etc. It also fills the original code region with junk instructions (Step ⑦).

This is an overview of our approach. We describe the implementation of Dsvmp in more details in the following sections.

V. Dsvmp Scheduling Structure

The Dsvmp VM scheduler uses multiple dispatchers to determine which bytecode instruction should be interpreted at given time. A unique design of Dsvmp is that the dispatcher used to schedule bytecodes will be dynamically changed at execution time. To further increase the diversity of program behaviour, Dsvmp also uses multiple bytecode instruction sets and bytecode handlers.

A. Multiple bytecode handlers

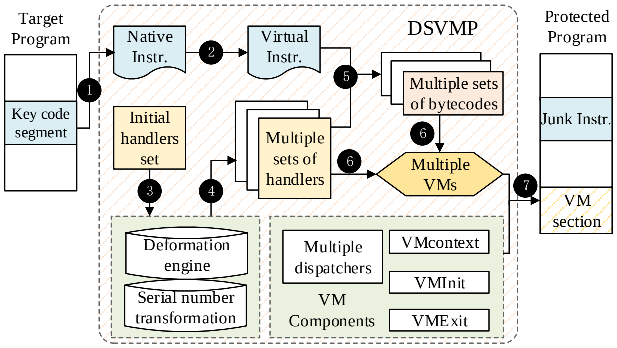


Figure 2: Offline code protection process. Dsvmp takes in a program binary. For each protected code region, it translates native instructions into bytecodes. Next, it generates multiple bytecode handlers that are semantically equivalent but implemented in different ways. It then generates the corresponding driver-data and multiple VMs. Finally, the generated VMs and associated components will be inserted into the program binary and fills the original code region with junk instructions.

```
1  lods byte/word/dword ptr ds:[esi]
2  ... ...
3  push eax
4  rdtsc                       ;-------------------------
5  mov ecx,2
6  div ecx                     ;structure control unit
7  cmp edx,0
8  jz label                    ;-------------------------
9  lods dword ptr ds:[esi]
10 ... ...                     ; to the next handler
11 add dword ptr ds:[edi+48],eax
12 jmp dword ptr ds:[edi+48]
13 label: push ebx             ;-------------------------
14 div bl
15 movzx eax,AH                ; return to a dispatcher
16 add eax,9dH
```

Figure 3: Each bytecode handler has a control unit that randomly determines whether the control after exiting the handler should be given to a dispatcher or an alternative bytecode handler.

In classical VM-based code obfuscation, a single dispatcher is responsible for fetching a bytecode instruction and determining which bytecode handler should be used to decode the bytecode. Because each bytecode instruction is decoded by a fixed handler, an adversary can easily work out the mapping of a bytecode instruction and its handler. From the mapping, the adversary can correlate the native machine code to each bytecode to analyze the program behavior.

To overcome this issue, for each bytecode handler, we create a number of alternative implementations which all produce an equivalent output for the same input. The alternative implementations, however, are implemented in different ways using e.g. different algorithms, data structures or obfuscation methods.

We insert a control unit at the end of each bytecode handler. Before exiting a bytecode handler, the control unit randomly determines whether the control should be given to a dispatcher or another handler. Figure 3 shows an example of a Dsvmp bytecode handler's control unit. The control unit (lines 4-8) randomly determines to execute the code at line 9 or line 13. At line 9 , the "`lods`" (a load operand in the x86 assembly) instruction fetch an offset value to calculate the address of an alternative bytecode handler. By contrast, the instruction at line 13 will return to a dispatcher randomly.

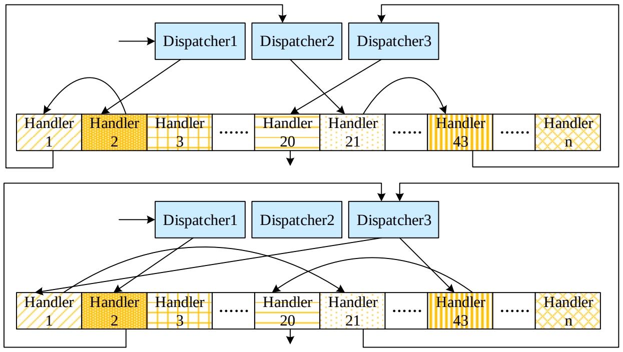


Figure 4: Using multiple dispatchers and insert a control unit to handlers increases the diversity of program executions. In this example, the type of handlers and the order of them are called are different across execution runs.

B. Multiple bytecode instruction sets and dispatchers

The bytecode instruction set determines the execution order of handlers. Compared to a single bytecode instruction set, multiple bytecode instruction sets provide stronger protection because the execution path of handlers will be more dynamic. Hence, DSVMP uses multiple bytecode instruction sets.

Our current implementation provides two bytecode instruction sets for each VM, *DriverData1* and *DriverData2*. The *DriverData1* is a standard bytecode instruction sets where each bytecode considers of the handler's serial number (a ID indicates which handler should use to interpret the virtual instruction) and the operand. *DriverData2* has a different format compared to *DriverData1*. The first data of *DriverData2* is the handler's serial number, The rest of *DriverData2* include the offset value between two adjacent handlers (for example, handler21 and handler43 in Figure 4) and the operand. Recall that a control unit is inserted to the end of each handler. Before exiting the handler, if the control unit chooses to execute the next handler, it will fetch the corresponding offset value from *DriverData2*. These bytecodes are all encrypted in our system.

DSVMP also provides multiple dispatchers to further increase the diversity of program execution. As an example, considering Figure 4 that shows two possible program execution using three dispatchers. As can be seen from the diagram, the handler also can schedule another handler and the type of handlers to be invoked and he order they are called are different in two different execution runs. Therefore, knowledge about the program control flow extracted from the first run does not apply to the second one.

VI. MULTIPLE VMS

In contrast to classical VM-based obfuscation approaches that uses a single VM (SVM), DSVMP uses multiple VMs. Multiple VMs offer different sets handlers and bytecode instruction sets. Under such settings, bytecode instructions can be scheduled by different VMs and a bytecode instruction can be interpreted by more than one handler. Therefore, there will be more than one mapping from a bytecode instruction to handlers. Together with the multiple bytecode instruction sets, multiple VMs further increase the diversity and uncertainly of program execution.

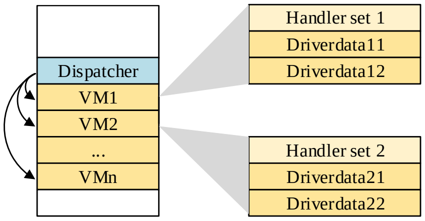


Figure 5: The structure of multiple VMs. Each VM has one set of unique handlers and two sets of bytecode instructions, *DriverDataSetN1* and *DriverDataSetN2*.

```
STARTSDK
00401036 mov eax , ebx
00401038 sub eax , 03
ENDSDK
```

Figure 6: Example assembly code snippet for a code region to be protected.

To schedule the multiple VMs, we need to alter the dispatcher structure. The dispatcher will need to determine which VM to use at runtime. To do so, we first calculate the address offet between the current VM and the VM to be used. We then store the value of the register `ESI` (a register points to the address of the current bytecode instruction) and change the pointer of the new bytecode address according to the offset value. Figure 5 shows the multiple VM structure. The VM, the set of bytecode handlers and bytecode instructions will be randomly switched across different code regions in both a single execution and across different program runs.

VII. EXAMPLE

We use the x86 code snippet shown in Figure 6 as an example to illustrate how DSVMP operates. `STARTSDK` and `ENDSDK` are used to mark the begin and end of the code region respectively, and 00401036 and 00401038 are the address the two assembly instructions.

A. Process of protection

Firstly, DSVMP automatically inserts two additional instructions ("`push 0x40103b`" and "`ret`") after two key instructions in order to jump back to execute the native code after the protected code region. It then converts the native instructions to virtual instructions according to a translation convention. The resulted virtual instructions is given in Table I. DSVMP's bytecode instructions are based on a stack machine model. Here the `load` instruction is used to push operands into the stack, and the `store` instruction is used to pop results out from the stack and store the result to the virtual context (VMContext).

After translating the native code to virtual instructions, we use the deformation engine to transform the initial bytecode handlers set. For this example, we generate two sets of bytecode handlers which are semantically equivalent but are implemented in different ways. We also randomly shuffle the serial numbers of these handlers, resulting in two new sets of handlers: `HAS1` and `HAS2`. Each set of bytecode handlers is associated with two bytecode instruction sets:

Table I: Generated virtual instructions for the example shown in Figure 6.

	Instr.1	Instr.2	Instr.3	Instr.4
NI	`mov eax, ebx`	`sub eax, 0x03`	`push 0x40103b`	`ret`

VI	`move 0x08`
	`load`
	`move 0x04`
	`store`
		`move 0x04`
		`load`
		`load 0x03`
		`sub`
		`store`
		`move 0x04`
		`store`
			`load 0x40103b`	`ret`

Notes: In the table, "NI" means the native x86 instructions, and "VI" means the virtual instructions. Add "Instr.3" and "Instr.4" in order to jump back to execute the native code after the protected code region.

DriverDataSet11 and *DriverDataSet12* for `HSA1` and *DriverDataSet21* and *DriverDataSet22* for `HSA2`. The resulted program is illustrated in Figure 7. We store the virtual instructions in the bytecode format.

We also encrypt the resulted bytecode instructions using different keys for different sets of handlers. For example, *DriverDataSet11* and *DriverDataSet12* will be encrypted using one key, and *DriverDataSet21* and *DriverDataSet22* will be encrypted using another key. We fill the code segment to be protected with junk instructions. Finally, we create a new code section attached to the end of the target program. The new code section contains the implementation of the handlers, different sets of bytecode instructions, dispatchers and other VM components such `VMContext` and routines such as `VMInit` (used to initialize the VM) and `VMExit` (use for cleanup before exiting the VM).

B. Runtime execution

Runtime execution of the protected code region is illustrated in Figure 7, which follows a number of steps:

- **Step1:** The entry of the protected code segment contains an "`jmp VMInit`" instruction. This transfers the control to the VM initialization routine, `VMInit`, which saves the host context and initializes the virtual context. The initialized routine also selects a VM to use. In this example, we assume we use two VMs and `VM2` is chosen at the beginning.
- **Step2:** Next, a dispatcher starts working. It fetches a bytecode from the *DriverDataSet21*. After decoding, the dispatcher gets an address of "`6a`". It then jumps execute "`0x6aHandler`", the next bytecode "`07`" is its operand.
- **Step3:** A control unit will be executed (see Section V-A) before exiting the "`0x6aHandler`". The control unit randomly selects to execute another handler, "`0x85Handler`", or return the controler to the dispatcher. If it chooses to return to the dispatcher, the program execution moves to Step 5.
- **Step4:** Assume the control unit decides to execute handler, "`0x85Handler`". It will then fetch a bytecode from *DriverDataSet22*, decoding it and getting the offset address of "`0x85Handler`" . Using the offset, the control unit will jump to `0x85Handler`. After executing the handler, the program execution moves to Step 3.
- **Step5:** If the control unit chooses to return the control to a dispatcher, it will randomly select a dispatcher to continue the execution. This moves to Step 6.

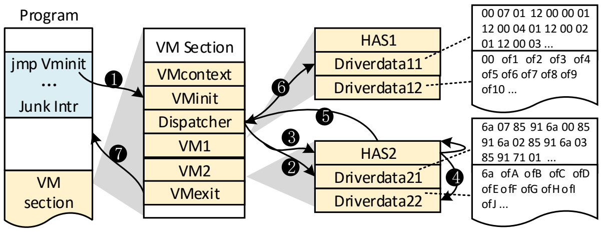


Figure 7: The execution process of the protected program. Here each VM has two sets of bytecode instructions and one set of handlers.

- **Step6:** The selected dispatcher randomly selects one VM to use. The dispatcher fetches a bytecode instruction from this VM, decoding the instruction to get the handler serial number. The handler serial number tells which handler to use to translate the bytecode to native machine code. It then jumps to execute the handler. After executing the handler, the program execution moves to Step 3.
- **Step7:** Step 3 and Step 4 are iterated until all the bytecodes get executed. The finally step is to invoke the `VMExit` to restore the native context and to continue executing the target program.

VIII. Security Strength Analysis

This section analyzes the security strength provided by DSVMP. We first analyze the number of possible execution paths. Then we discuss the diversity of code structures.

A. Program execution paths

Recall that our design goal is to increase the diversity of program execution, so that in different runts the protected region will not follow a single execution path across runs. In this analysis, we assume there are 10 different dispatchers. This number matches the current implementation of DSVMP. We use the example presented in Section V-B as a case study. In this example, *DriverDataSet11* and *DriverDataSet21* each has 103 bytes of data. They contain a total of 78 handler serial numbers. In this analysis, we exclude the last handler because of it is used to exit the VM. This leave us 77 handlers where each handler can lead to 11 different execution paths. This is because at the end of executing each handler, a control unit will determine whether the control should be given to another handler or one of the 10 dispatchers (see Section V-A) – 11 possibilities in total.

In combination, these options give 11^{77} possible execution paths for each protected code region. Therefore, the probability, p, for a protected code region to follow the same execution path across different runs is $p = \frac{1}{11^{77}}$, a very small number. Bear in mind that so far we have assumed that the protection scheme uses just one VM. The multi-VM strategy employed by DSVMP further increases the number of possible execution paths. In fact, the more dispatchers and VMs are, the greater number of possible execution paths will be. The current DSVMP implementation provides five different VMs. Together with the multiple dispatchers and bytecode instruction straggles, for the setting used in this

Table II: The relevant information about the program.

Basic info of program		Info of protected-software				
prog.	key code	prog.	Node Num	Branch Num	$\sum_{i=0}^{i<n} DR(i)$	$\sum_{i=0}^{i<n} DF(i)$

A	`mov eax,ebx`
	`sub eax,03`
		A’	23	5	46	18

B	`pop eax`
	`add eax,ebx`
		B’	48	9	96	36

Notes: In the table, the number of `n` which in $\sum_{i=0}^{i<n} DR(i)$ and $\sum_{i=0}^{i<n} DF(i)$ are equal to the `NodeNum`.

section, DSVMP gives a single code region 11^{385} possible execution paths. Given the massive number of choices, it will be rare for a protected code region to take the same execution path across different runs.

B. Code structures

To prevent an adversary reuses knowledge obtained from other software to perform attacks, we would like applications protected by DSVMP exhibit distinct code structures. In other words, we would like programs after code obfuscation to be as much dissimilar as possible in terms of code structures.

Blietz *et al.* [15] proposed a method to measure the similarity of program structures, using control flow information such as the number of branches and back blocks, the nesting level of the code etc. We draw lessons from this method to analyze code structures for programs protected using DSVMP. We use a number of metrics to describe program code structures. These metrics are:

- `NodeNum`: the number of basic blocks of the protected region.
- `BranchNum`: the number of basic blocks where the last instruction is a conditional jump instruction.
- $DR(Vi)$: the number of in and out instructions for the basic block, *Vi*. This metric is defined as $DR(Vi) = D_{in}(Vi) + D_{out}(Vi)$ where $D_{out}\,(Vi)$ refers to the out-degree and $D_{in}\,(Vi)$ refers to the in-degree and they mean the number of arcs that start or end at Vi.
- $DF(Vi)$: the data flow relationship of basic block, *Vi*. This is used to measure the frequency of *Vi*’s information exchange. It is defined as $DF\,(Vi)\; = Flow_{in}(Vi) + Flow_{out}(Vi)$, where $Flow_{in}$ is the number of reading instruction in *Vi* and $Flow_{out}$ is the number of writing instruction in *Vi*.

Table II gives two examples of code regions to be protected. These are two simple code snippets and without code obfuscation, these two examples have very similar structures because all of then with just one basic block and no branches. Transforming the code regions using DSVMP, we obtain different metric values for both code regions, which indicate the transformed code segments have distinct structures. We use the following formula to quantify the code structure information, X after code obfuscation.

$$\begin{aligned}&SInfor_X = NodeNum_X + BranchNum_X\\&+\sum_{i=0}^{i<n}(DR(i)+DF(i))\end{aligned}$$

Applying this formula for the transformed code segments, A’ and B’, listed in Table II, we get :

$$\begin{aligned}&SInfor_{A'} = NodeNum_{A'} + BranchNum_{A'}\\&+\sum_{i=0}^{i<n}(DR(i)+DF(i))\\&= 23+5+(46+18)\\&= 92\\&SInfor_{B'} = NodeNum_{B'} + BranchNum_{B'}\\&+\sum_{i=0}^{i<m}(DR(i)+DF(i))\\&= 48+9+(96+36)\\&= 189\end{aligned}$$

Among them, $n = NodeNum_{A'}$ and $m = NodeNum_{B'}$. From $SInfor_{A'}$ and $SInfor_{B'}$, we can calculate the similarity of code structure, $SDiff$, for A’ and B’ as:

$$SDiff = \frac{|SInfor_{A'} - SInfor_{B'}|}{SInfor_{A'} + SInfor_{B'}} = \frac{97}{281} = 34.5\%$$

Thus it can be seen the code structure similarity between two A’ and B’ is 34.5%. This example shows that DSVMP can significantly increase the dissimilarity of code structures even for simple code segments. We also observe that the similarity between transformed code regions drops significantly as the complexity of original code segments increases.

IX. PERFORMANCE EVALUATION

A. Attack Experiment

In our experiments, there are three VMs and five dispatchers for DSVMP. We uses IDA [11] to debug the program to obtain the control flow graph of the protected code region. However, for those basic block that ends with an indirect jump instruction, IDA can not get the target address automatically. This means that the control flow graph is not complete, and its maybe lack of some key connection between the basic blocks. We can take a lot of time to manually connect those basic blocks to get a full control flow graph, but the control flow has changed when we try to process the program again. Previous attack experience are useless, we can only take time again to repeat the above work in current state.

DSVMP also uses OllyDbg [12] to perform the dynamic debugging, and then locate the dispatcher. It uses the x86 register `ESI` as the program counter for bytecode instruction. Further, we uses `ESI` to track the movement of tainted data and find that the bytecodes will be decrypted and stored in register `EAX`. This value is the serial number of bytecode handler which should be executed. We can use these Values to analyze the execution order of handlers, so repeat the above operation and collect all those values that scheduled by each dispatcher. All these data, however, still cannot restore the execution logic of handlers, because these bytecodes from multiple VMs, the same data may represent different meanings. In addition, the handlers that we collected by dispatcher are not all of them, because in DSVMP some handlers probably scheduled by other handlers (see Section V). What counts is DSVMP has the different bytecode handler scheduling for different runs, which makes the dynamic debugging more difficult.

Table III: Information of the bechmarks.

prog.	Size(KB)	Func. to protect	Instr. Protect	Instr. Executed
md5	11	`Transform`	563	6869163
aescrypt	142	`encrypt-stream`	1045	5502747
bcrypt	68	`Blowfish-Encrypt`	54	43945017
gzip	56	`deflate`	154	35877278

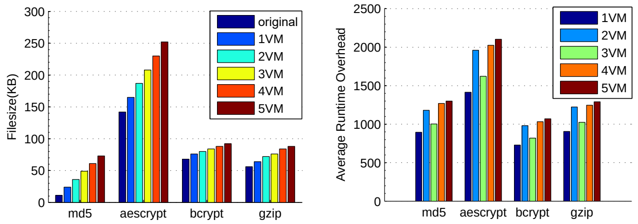
B. Evaluation platform and benchmarks

We evaluated DSVMP on a PC with an 3.0 GHz Intel Core^{TM} 2 Duo processor and 4GB of RAM. The PC runs the Windows 7 operating system. We evaluated our approach using four widely use applications: `md5` [16], `aescrypt` [17], `bcrypt` [18] and `gzip` [19]. We used these applications to process a test image file. The size of the file is 763 KB. Table III gives information of the protected code regions for each benchmark. The 3rd column of the table gives the function to be projected and the 4th column shows the number of instructions of the function. Finally, the number of instructions got executed with the functions while processing the test file is shown in the last column of the table.

C. Code Size and Runtime Overhead

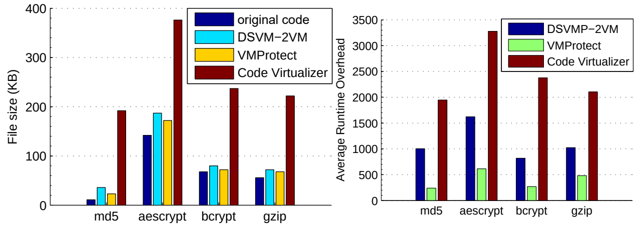
Code size: For each target benchmark, we applied DSVMP to the target function and repeated the process for five times. For each protection run, we used a different number of VMs. Figure 8 (a) shows how the DSVMP multi-VM scheme affects the code size. As described before, each VM has two bytecode instruction sets and one set of handlers, the code size of the protected program grows as the number of VM increases. Moreover, there is a strong correlation between the code size of and the number of protected instructions. This explains why `aescrypt` has the fastest increase of code size as it has the largest number of protected instructions (see Table III). For the same reason, the code size of `bcrypt` grows slower than other programs, as this benchmark has the least number of protected instructions.

Runtime overhead: To evaluate the runtime overhead of DSVMP, we used each benchmark to process the file. We repeated the process for 10 times and report the average runtime overhead per benchmark. The results are shown in Figure 8 (b). As can be seen from this diagram, the runtime overhead increases as the number of VM used increases. The only exception is that the 3-VM configuration (3VM) has a lower overhead than a 2-VM one. One possible reason is that the implementation strategy we used for multi-VM protection. FIX:Here we are still trying to solve it. Furthermore, `aescrypt` has a much higher runtime overhead than other benchmarks. The `encrypt-stream` function of `aescrypt` is more complex and thus has a higher number of virtual instructions compared to other benchmarks. As such, it takes longer to execute the obfuscated code for this program compared to other benchmarks.



(a) (b)

Figure 8: (a) The impact of code sizes for configurations with a different number of VMs. (b) The average runtime overhead per instruction with different VMs.



(a) (b)

Figure 9: (a) The comparison of impact on file size with VMProtect and Code Virtualizer. (b) The comparison of average runtime overhead per dynamically executed critical instruction with VMProtect and Code Virtualizer.

D. Comparisons with state-of-the-arts

We also compared DSVMP against two commercial VM protection systems, Code Virtualizer (CV) [2] and VMProtect [3], in terms of code sizes and runtime overhead. We use the FISH32 (White) VM from the 24 customized VMs of CV to perform code protection because this VM has moderate runtime overhead. We used a configuration of two VMs for DSVMP, DSVMP-2VM, in this experiment.

Code size: Figure 9 (a) shows the impact on code size of three VM-based protection systems. The code size of programs protected under CV grows faster than the other two schemes. However, the resulted code size of CV is relatively stable across benchmarks. On the other hand, the code sizes of DSVMP and VMProtect are comparable and smaller than CV.

Runtime overhead: Figure 9 (b) shows the average runtime overhead of the three schemes. Code protected under CV has the most expensive runtime overhead, which on average is 2x higher than DSVMP. Among the three schemes, VMProtect has the smallest overhead while DSVMP has a slightly higher overhead compared to VMProtect. Consider that DSVMP provides stronger protection with a diverse set of handlers, dispatchers and VMs, we argue that the modest increase in runtime overhead is acceptable.

X. Related Work

Early work on the binary code protection relies on simple encryption and obfuscation methods, but they are vulnerable to the sophisticated, diversified attacks developed over the past years. Traditionally, techniques like junk instructions [20], packers [21], [22], are used to protect software against attacks based on disassembly and static analysis. There are also other code protection techniques like code obfuscation [23], control flow and data flow obfuscation [24], [25], [26], all aim to obfuscate the semantic and logical information of the target program. In practice, these approaches are often used in combination to provide stronger protection. DSVMP also leverages some of the code obfuscation techniques developed in the past for code protection.

There is a growing interest in using code virtualization to protect software from malicious reverse engineering. Fang *et al.* [4] proposed a protection scheme based on multi-stage code obfuscation. Their approach iteratively transforms the critical code region several times with different interpretation methods to improve security. Yang *et al.* [5] presented a nested virtual machine for code protection. Using their approach, an adversary would have to fully reverse engineer a layer of the interpreter before moving to the next layer, which increases the cost of attacks. Averbuch *et al.* [27] introduces an encryption and decryption technology on the basis of VM-based protection. This approach uses the AES algorithm and a customize encryption key to encrypt the virtual instructions. During runtime, the VM will decrypt the virtual instruction and then dispatch a handler to interpret the virtual instructions. Wang *et al.* [6] proposed a protection scheme to increase the time diversity of protected code regions. This is achieved by constructing several equivalent but different forms of sub program execution paths, from which a path will be randomly selected to execute at runtime.

As a departure from prior work, DSVMP presents a dynamic scheduling structure to improve security for software. DSVMP has integrated several novel techniques to increase the diversity and uncertainly of program execution. These include using a control unit to diversify the execution path of bytecode handlers and using multiple VMs and dispatchers to randomly schedule instructions from multiple bytecode instruction sets. Integrating these techniques allows DSVMP to provide a more diverse program execution structure compared to prior work in the area. This richer set of diversity can better protect software against code reverse engineering [28].

XI. CONCLUSIONS

This paper has presented DSVMP, a novel VM-based code protection scheme. DSVMP uses a dynamic scheduling structure and multiple VMs to increase diversity of program execution. We have shown that code segments protected by DSVMP rarely follow the same execution path across different runs. The dynamic program execution brought by DSVMP forces the attacker to have to use many trail runs to uncover the implementation of the protected code region. As such, DSVMP significantly increases the overhead and effort involved in code reverse engineering. We have evaluated DSVMP using four real world applications and compared it to two state-of-the-art VM-based code protection schemes. Our experimental results show that DSVMP provide stronger protection with comparable overhead of runtime and code size.

ACKNOWLEDGMENT

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).

REFERENCES

[1] "Themida," http://www.oreans.com/themida.php.
[2] "Code virtualizer," http://www.oreans.com/codevirtualizer.php.
[3] "Vmprotect software. vmprotect," http://vmpsoft.com/.
[4] H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage binary code obfuscation using improved virtual machine," in *Information Security*. Springer, 2011, pp. 168–181.
[5] Y. Ming and H. Liusheng, "Software protection scheme via nested virtual machine," *Journal of Chinese Computer Systems*, vol. 32, no. 2, pp. 237–241, 2011.
[6] H. Wang, D. Fang, G. Li, N. An, X. Chen, and Y. Gu, "Tdvmp: Improved virtual machine-based software protection with time diversity," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, 2014.
[7] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, and Y. Gu, "Nislvmp: Improved virtual machine-based software protection," in *9th International Conference on Computational Intelligence and Security (CIS)*, 2013.
[8] N. Falliere, P. Fitzgerald, and E. Chien, "Inside the jaws of trojan," Clampi. Technical report, Symantec Corp, Tech. Rep., 2009.
[9] C. Collberg, "The case for dynamic digital asset protection techniques," *Department of Computer Science, University of Arizona*, pp. 1–5, 2011.
[10] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on Software Engineering*, 2002.
[11] "Ida pro," https://www.hex-rays.com/index.shtml.
[12] "Ollydbg," http://www.ollydbg.de/.
[13] "Sysinternals suite," https://technet.microsoft.com/en-us/sysinternals/bb842062.
[14] R. Rolles, "Unpacking virtualization obfuscators," in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.
[15] B. Blietz and A. Tyagi, "Software tamper resistance through dynamic program monitoring," in *Digital Rights Management. Technologies, Issues, Challenges and Systems*, 2006.
[16] "Md5," http://www.fourmilab.ch/md5/.
[17] "Aescrypt," https://www.aescrypt.com/download/.
[18] "bcrypt-blowfish file encryption," http://sourceforge.net/projects/bcrypt/.
[19] "gzip," http://www.gzip.org/.
[20] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.
[21] "Execryptor," http://www.strongbit.com/execryptor.asp.
[22] "Upx," http://upx.sourceforge.net/.
[23] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.
[24] C. Liem, Y. X. Gu, and H. Johnson, "A compiler-based infrastructure for software-protection," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, 2008.
[25] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *Proceedings of the 5th ACM workshop on Digital rights management*, 2005.
[26] V. Balachandran, N. W. Keong, and S. Emmanuel, "Function level control flow obfuscation for software security," in *Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2014.
[27] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "An efficient vm-based software protection," in *5th International Conference on Network and System Security (NSS)*, 2011.
[28] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.