Enhance Virtual-Machine-Based Code Obfuscation Security Through Dynamic Bytecode Scheduling [1]

Kaiyuan Kuang[a], Zhanyong Tang[a,*], Dingyi Fang[a], Xiaojiang Chen[a], Zheng Wang[b]

[a] *School of Information Science and Technology, Northwest University, China.*
[b] *School of Computing and Communications, Lancaster University, UK*

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 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.

Keywords: Code virtualization, Code Obfuscation, Dynamic cumulative attack

[1] Extension of Conference Paper: a preliminary version of this article entitled "Exploiting Dynamic Scheduling for VM-Based Code Obfuscation" by K. Kuang, Z. Tang and Z. Wang et el. appeared in the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications(TrustCom), 2016. The extended version makes the following several additional contributions over the conference paper: (1)it provides more detailed description of the background and threat models (Section 2 and 3); (2)it describes how the virtual interpreter dynamically changes the execution path at runtime with an algorithmic model (Section 6.2 and Algorithm 1); (3)it provides new experimental result on space-time costs of DSVMP (Section 9.2); (4)it adds a new experiment to evaluate the protected structure diversity of the protected program at runtime (Section 9.3, Figure 10, 11, 12); (5)it includes new results to compare our approach against two commercial VM protection systems, Code Virtualizer and VMProtect (Section 9.4).

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

Email addresses: kky@stumail.nwu.edu.cn (Kaiyuan Kuang), zytang@nwu.edu.cn (Zhanyong Tang),

1. 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 based on 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.

dyf@nwu.edu.cn (Dingyi Fang), xjchen@nwu.edu.cn (Xiaojiang Chen), z.wang@lancaster.ac.uk (Zheng Wang)

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 algorithms 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.

The rest of this paper is organized as follows. Section 2 introduces the principle of classical VM-based code obfuscation techniques and cumulative attacks scenario. Section 3 describes the VM reverse attacking approach. Section 4 gives an overview of DSVMP , which is followed by a detailed description of the design in Section 5 and 6. Section 7 uses a case study to demonstrate protection scheme provided by DSVMP . Evaluation results are presented in Sections 8 and 9 before we discuss the related work in Section 10. Finally, Section 11 presents our work conclusions.

2. Background

VM-based code obfuscation. VM protection system extracts the critical code from the target program and disassembled into native instructions, then transforms native instructions to virtual instructions. The following, virtual instructions are encoded into bytecode program

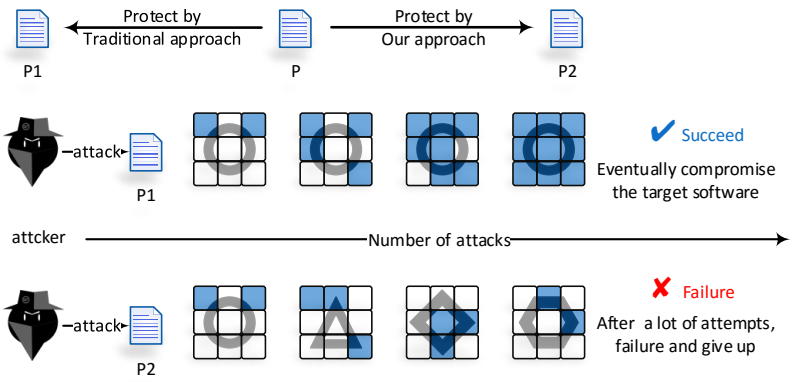

Figure 1: Diversity affects effectiveness of the attack. In this example, a dark small square represents reusable attacking knowledge. Diverse program execution increases the difficulty of attacks.

that can only be interpreted by the special interpreter of the system itself. Virtual instructions are used to emulate native instructions, which means that the virtual instructions set need to be able to fully implement the semantics of native instructions. Classical VM instructions design are based on a stack machine model where computation is performed using stack operations like push and pop.

After the code virtualized 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. The VM section contains a bytecode program and some important VM components. When entering the VM, component VMinit saves the native context and initializes the virtual context. The 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, which consisting of a number of important virtual registers. At the heart of the VM is an interpreter with two components: Dispatcher and handlers set. The dispatcher fetches a bytscode from bytecode program and decodes it, then dispatches a handler to interpret it into the native code. This process will iterate until all the bytecodes are interpreted. When exiting the VM, component VMExit restore the native context, and the program jumps back to the native code following the critical code segment and continues to execute the rest of the program 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.

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.

3. The Attack Model

The classical approach to reverse engineer a VM-protected program typically follows three steps [8, 10].

The first step is to reverse engineer the essential components of a VM interpreter, and the purpose of this step is to get the location and interaction information of each component of the virtual interpreter. 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 and work out the semantics of the bytecode instructions. The third step is to use knowledge obtained in the first two steps to recover the original logical implementation of the target program, then remove the redundant information and restoring a program that is similar to the original program.

This type of reverse approach takes certain analysis time and requires attacker to have a certain understanding of the principle of code virtualization. However, a skilled attacker is able to use the method of cumulative attacks, then reuse 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 that the attacker has the necessary tools and skills to implement the above attacks. we assume the adversary holds an executable binary of the target software and can run the program in a malicious host environment [11]. 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" [12], "Ollydbg" [13] and "Sysinternals suite" [14]. 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.

4. 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 5) and multiple VMs (Section 6) 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 ❶), which will then be mapped into a set of virtual instructions (Step ❷).

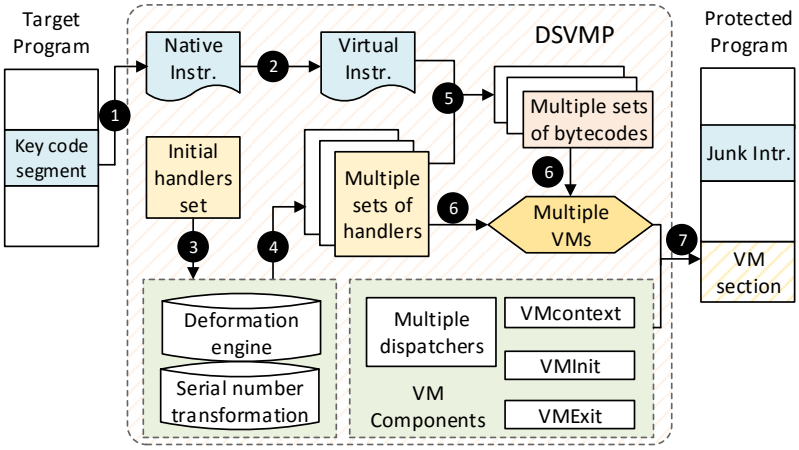

Figure 2: Offline code protection process. Dsvmp takes in a program binary. For each protected code region, it translates native instructions into bytecodes. Next, it generates multiple bytecode handlers that are semantically equivalent but implemented in different ways. It then generates the corresponding 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.

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 ❸). 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 ❹). 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 5.2) (Step ❺). Subsequently, Dsvmp constructs multiple VMs, where each VM contains one set of handlers and two sets of bytecodes (Step ❻).

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 ❼).

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

5. 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 bytecode handlers 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.

```
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.

5.1. Multiple bytecode handlers

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 interpret the bytecode. Because each bytecode instruction is decoded by a fixed handler set, 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 and logic structure.

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 and jump to execute it. By contrast, the instruction at line 13 will return to a dispatcher randomly.

5.2. Special bytecodes program and multiple dispatchers

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

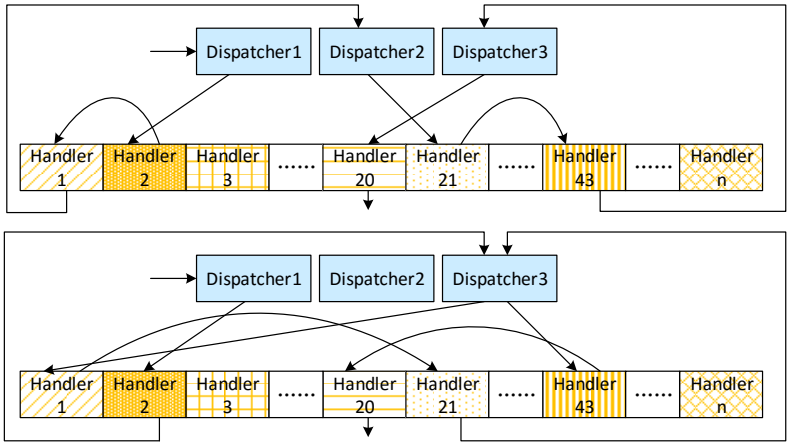

Figure 4: Using multiple dispatchers and insert a control unit to handlers increases the diversity of program executions. In this example, the type of handlers and the order of them are called are different across execution runs.

Our current implementation provides two bytecode instruction sets for each VM, *DriverData1* and *DriverData2*. The *DriverData1* is a standard bytecode program where each bytecode considers of the virtual instruction's opcode (a ID indicates which handler should use to interpret the virtual instruction) and their operand. *DriverData2* has a different format compared to *DriverData1*. The first data of *DriverData2* is the handler's ID, The rest of *DriverData2* include the offset value between two adjacent handlers (for example, handler21 and handler43 in Figure 4) and their 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*.

DSVMP also provides multiple dispatchers to further increase the diversity of program execution. Finally, scheduling structure of the DSVMP with a single VM as an example, considering Figure 4 that shows two possible program execution using three dispatchers and control unit in a single VM. As can be seen from the diagram, a handler can not only be scheduled through a dispatcher, but also can be scheduled by 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.

6. 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 from different VMs and a virtual 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 above multiple scheduling approach, multiple VMs can further increase the diversity and uncertainly of program execution.

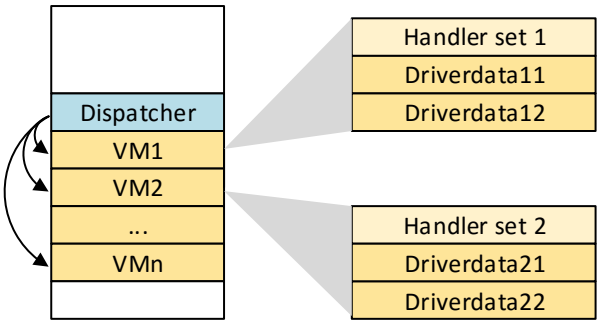

Figure 5: The structure of multiple VMs. Each VM has one set of unique handlers and two sets of bytecode instructions, *DriverDataSetN1* and *DriverDataSetN2*.

6.1. Multiple VMs switch

We can determine the number of VM by ourselves when protecting target program. And then, according to the number of VM, the handler set is confused with the corresponding number of times and re allocation their opcode, then get multiple sets of equivalent but different forms of handlers. To review the section 5.2, for each set of handler sets, virtual instructions of the target code will be translated into two groups of special bytecode program. These bytecodes in different VM with semantics diversity, so there will be more than one mapping from a bytecodes to handlers.

To schedule the multiple VMs, similar to the handler, we also add a switcher to alter the dispatcher structure. The dispatcher will has the function 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.

6.2. Scheduling process of Dsvmp *with multi-VMs*

VM-based protection system uses its own virtual interpreter to interpret the special bytecode program into the native code (by scheduling various handlers to implement discrete sub operations) to realize the function of the original target code. We add the special control unit in the execution process of virtual interpreter, its main works are as follows:

- structure control unit determines whether the control should be given to a dispatcher or another handler randomly.
- multi-VM switcher determines which VM to use at runtime randomly.

In multiple VMs, all of the handler sets are semantic equivalence, so Dsvmp can ensure that the function of the target program is correct after randomly selecting. The collaboration of different components achieves the ultimate goal of the strong protection of the original

Algorithm 1 Virtual Interpreter's Work Flow

```
VMInit
Switcher selects a VM randomly
Fetch a bytecode from DriverData1 in current VM
while bytecode ≠ ∅ do
    Decoding the bytecode
    JMP handler to interpreter it
    Control unit randomly transfer control
    if Choose the dispatcher then
        Select a dispatcher randomly
        Switcher selects a VM randomly
        Fetch the next bytecode from DriverData1 in current VM
    else {Choose the next handler}
        Fetch the next bytecode from DriverData2 in current VM
    end if
end while
VMExit
```

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

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

target program, and the pseudo description of the DSVMP virtual interpreter shown as follows:

Algorithm 1 gives the work flow of virtual interpreter. When the target program entering the VM, virtual interpreter will fetch the bytecode from *DriverData* and dispatch the handler to interpret it. When all the bytecodes are interpreted, the function of original target code also will be implemented. In this process, the role of the switcher is randomly switches the VM at runtime, and the control unit will randomly change the scheduling and execution structure of current VM at runtime. So our DSVMP can exploit the dynamic scheduling for VM-based protection at runtime. Such run time diversity can effectively increase the difficulty of the reverse analysis and resist the cumulative attacks.

7. 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.

7.1. Process of protection

Firstly, DSVMP extracts the critical code from the target program that disassembled into native instruction, here will automatically insert 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, as showed in figure 6. It then converts the native instructions to virtual instructions according to a translation convention. The resulted virtual instructions is given in Table 1. 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).

Table 1: 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" indicates the native x86 instructions, and "VI" donates the virtual instructions. Here, our system inserts "Instr.3" and "Instr.4" in order to jump back to execute the native code after returning from the protected code region.

After translating the native code to virtual instructions, we use the deformation engine to transform the initial bytecode handlers set. For this example, our system with 2 VMs configurations, so we generate two sets of bytecode handlers which are semantically equivalent but are implemented in different ways. Then we 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: *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, and will fill the critical code segment to be protected with junk instructions.

Finally, DSVMP 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).

7.2. 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 native host context and initializes the virtual context.

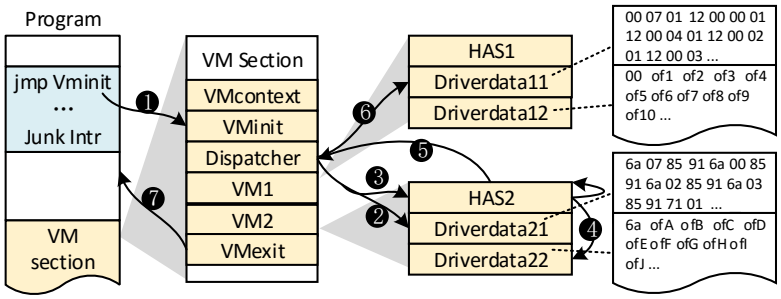

Figure 7: The execution process of the protected program. Here each VM has two sets of bytecode instructions and one set of handlers.

- **Step2:** Next, a dispatcher starts working. It randomly selects a VM, we assume that `VM2` is chosen at beginning, and then it fetches a bytecode from the *DriverDataSet21*. After decoding, the dispatcher gets a bytecode "`6a`". It then jumps to execute "`0x6aHandler`", and the next bytecode "`07`" is its operand.

- **Step3:** A control unit execute (see Section 5.1) before exiting the "`0x6aHandler`". The control unit randomly selects to execute another handler, "`0x85Handler`", or return the controller to the dispatcher. If it chooses to return to the dispatcher, the program execution moves to Step 5.

- **Step4:** Assume that the control unit decides to direct execute the next 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 execute "`0x85Handler`". After executing the handler, the program execution moves to Step 3.

- **Step5:** If the control unit chooses to return the controller to a dispatcher, it will randomly select a dispatcher to continue the execution. Then, the program moves to Step 6.

- **Step6:** The selected dispatcher randomly selects one VM to use. The dispatcher fetches a bytecode from this VM, decoding the bytecode to get the handler serial number. 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 rest of target program.

8. 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.

8.1. 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 7 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 5.1) – 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 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.

8.2. Code structures

To prevent an adversary from resuing 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 2: The relevant information about the program

Basic info of program		**Info of protected-software**				
program	**key code segment**	**program**	**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`.

Table 2 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.

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

Applying this formula for the transformed code segments, A' and B', listed in Table 2, we get :

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

where $n = NodeNum_{A'}$ and $m = NodeNum_{B'}$. From $SInfor_{A'}$ and $SInfor_{B'}$, we can calculate the similarity $SDiff$, for two code structure, 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.

9. Performance Evaluation

In this section, we evaluated DSVMP using four widely use application and compared it against two commercial VM-based protection systems, then present the evaluation results of DSVMP based on the experimental data in detail.

Table 3: Information of the bechmarks

program	Size(KB)	Instr. Total	Function to protect	Instr. Protect	Instr. Executed
md5	11	1357	`Transform()`	563	229141
aescrypt	142	9788	`encrypt-stream()`	1045	478297
bcrypt	68	3081	`Blowfish-Encryp()`	54	1050003
gzip	56	9837	`deflate()`	154	680037

Notes: The 3rd column shown the total number of target program instructions. The 4rd column of the table gives the function to be projected and the 5th column shows the number of instructions of the function. The number of instructions got executed with the critical functions while processing the test file, and shown in the last column of the table.

9.1. Evaluation Platform and Benchmarks

We evaluated DSVMP on a PC with an 3.0 GHz Intel Core^{TM} 3 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 text file. The size of the file is 26 KB. Table 3 gives some information of the protected code regions for each benchmark. The total number of target program instructions are shown in the 3rd column of the table. The 4rd column of the table gives the function to be projected and the 5th column shows the number of instructions of the function. Finally, we use the Intel Pin tools [20] to calculate the number of instructions got executed with the critical functions while processing the test file, and shown in the last column of the table.

9.2. Code Size and Runtime Overhead

Code size. For each target benchmark, we applied DSVMP to the target function and repeated the process for 8 times. For each protection run, we used a different number of VM configuration.

Figure 8 shows how the DSVMP multi-VM scheme affects the code size, and the number "0" represents the original target program. 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. Obviously, the increase in code size is regular, because the block of PE executables are usually follow a certain alignment value (such as, 4096 or 512) [21].

Moreover, we can find that 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 3). For the same reason, the code size of "`bcrypt`" grows slower than other programs, as this benchmark has the least number of protected instructions.

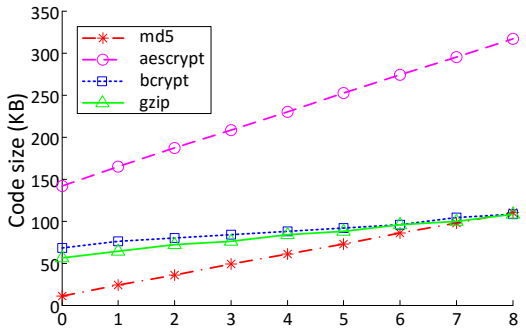

Figure 8: The impact of code sizes for DSVMP configurations with a different number of VMs. The number of horizontal axis is the configuration of the VMs, "0" is the original program.

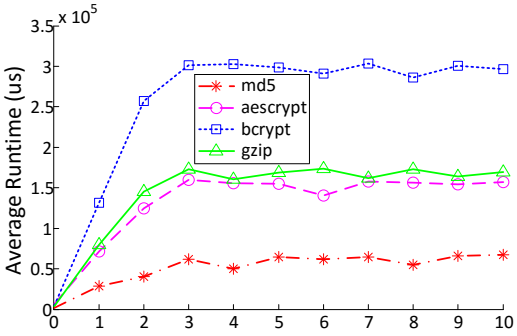

Figure 9: The average runtime of target benchmark when protected with different VMs. The number of horizontal axis is the configuration of the VMs, "0" is the original program.

Runtime overhead. To evaluate the runtime overhead of DSVMP , we used benchmark to process the test file. For each protected benchmark we repeated the process for 10 times and report the average runtime per benchmark.

The results are depicted in Figure 9, which shows that the majority of average runtime will tend to a stable range with the increase of the number of VMs. This phenomenon is inevitable, the increase in the number of VMs leads to a greater likelihood and diversity of handler's choice, when interpret the virtual instructions. It does not add a lot of extra operations to the core function implementation. The main reason for the impact of runtime overhead is the switching of multiple VMs and the execution difference of obfuscated handler. As the number of VMs increases, the frequency of VM switching tends to stabilize, so the impact of different VM on the time is different but tends to a stable range.

Besides, we found that the greater the execution number of the protected instruction at runtime will have a greater impact on runtime overhead. This is why "`bcrypt`" has a much higher runtime overhead than other benchmarks.

There is also a special phenomenon that compared to other protection strategies, 2-VM increase in running time overhead is not obvious. This is due to the fact that the random policy we are using does not frequently switch VM in a 2-VM configuration. Therefore, its impact on time overhead will not be obvious. We have done the relevant test experiments to validate it, see Section 9.3 for details.

Discussion. Comprehensively assess the experimental results of several target benchmarks in table 3. We find the temporal and spatial overhead of the target program is mainly affected by the size of the key code and the number of instructions at execution time. So we have reason to believe, if we only focus on protecting a few critical code regions, DSVMP will not cause too much impact on the temporal and spatial overhead of the target program.

9.3. Structural diversity

Multi-VM switching test. In order to verify the impact of multi-VM switching on program execution, we did a runtime tracking experiment. We only protected one instruction "`mov`

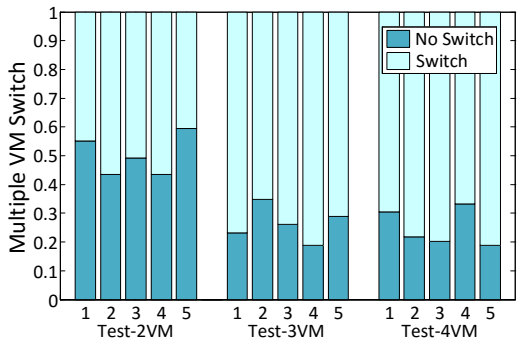

Figure 10: The probability distribution of multiple VMs switching for multiple runs. "Switch" is the number of times the VM switch, and "No Switch" is the opposite.

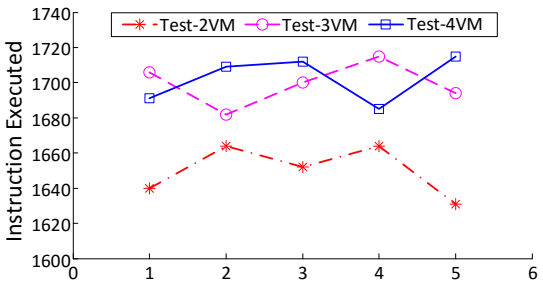

Figure 11: The number of instruction dynamic executions changes with different VM configurations. Here it is mainly affected by the Multiple VM switch.

`eax, 1234567`" for "`test.exe`"[2]. So as to reduce the complexity of the protected program as much as possible to facilitate the tracking procedures. We use three kinds of strategy (2-VM, 3-VM and 4-VM) to protect the target program, and then track the program execution 5 times and collect the relevant information.

Figure 10 shows the VM switching frequency information of "`test.exe`" at five actual execution. A total of 69 times handler scheduling are required to implement the test key instruction functions. We found that the frequency of VM switching is not high when the number of VMs is 2, unlike the 3-VM and 4-VM of the switching frequency has exceeded 75%. In particular, we removed handler obfuscation and used only one dispatcher in the protection process. This can minimize the impact of irrelevant factors, and then only the multi-VM switcher can affect the number of instruction executed. Through the experiment we found that the impact of 2-VM on the number of instructions got executed is less than 3-VM and 4-VM, as the figure 11 shows. In general, the number of instruction execution can be reflected to some extent the size of the execution time. So this can also explain why the 2-VM has a smaller effect on runtime overhead in figure 9.

Perform path runtime diversity. In the course of the above experiment, we also collected the ID of the VM where the handler was executed each time. Figure 12 shows the switching of the VM for five runs. The data in (a) and (b) are from the test program with 3-VM and 4-VM configurations, respectively.

As can be clearly seen in the figure, VMs are randomly selected for different runs, and the path of execution 5 times is completely different. As alluded to earlier (section 6), the set of handlers in each VM is confused with different obfuscation methods and is combined in an out-of-order manner. Therefore, the structure of the called Handler is different for the same phase of the different runs. Take 4-VM configuration test program as an example, a total of 69 Handlerd scheduling, regardless of the middle of the dispatcher changes in the process,

[2]A small test procedures, size of 3 KB, and its function is to pop up a confirmation box.

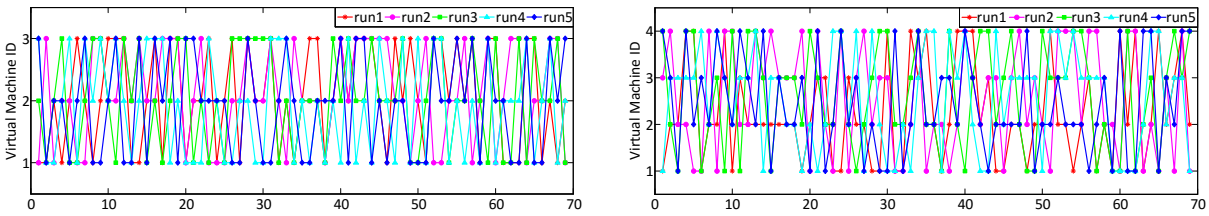

Figure 12: Dynamic VM switching during the protected program execution. The handler's scheduling order is plotted on the abscissa, starting at the origin and moving to the right. There are a total of 69 schedules. The value on the ordinate corresponds to VM's ID which is selected for each schedule. Lines of different colors show the true change of the VM in 5 executions. (a) is "Test-3VM", and (b) is"Test-4VM".

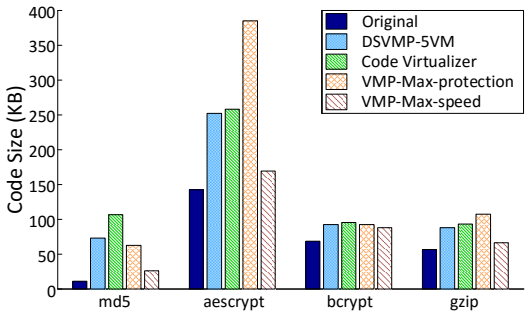

Figure 13: The comparison of impact on file size with VMProtect and Code Virtualizer.

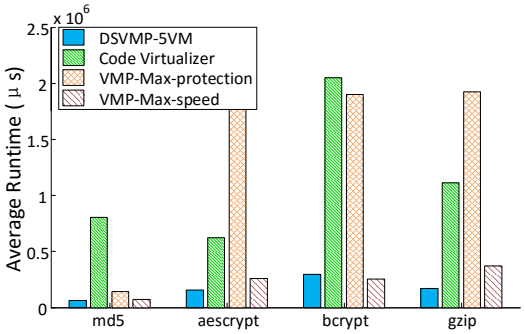

Figure 14: The comparison of average runtime overhead with VMProtect and Code Virtualizer.

only the Handler scheduling execution there is 4^{69} different execution path. This value is very large, and it will be great resistance to the adversary who use cumulative attacks.

Discussion. According to the experimental results, we can see, the execution path of a protected program is variable at runtime. Combined with various handler sets, the target program protected by DSVMP has temporal diversity [9]. Therefore, our approach can effectively resist the cumulative attacks.

9.4. Comparisons with state-of-the-arts

We also compared DSVMP against two commercial VM protection systems, Code Virtualizer (CV) [2] and VMProtect (VMP) [3], in terms of code sizes and runtime overhead. We adopt a custom protection scheme when using the CV to protect the target program, because such a CV has a moderate temporal and spatial overhead. In addition, we also adopt the VMP with two kinds of schemes to protect the target program, *Maximum-protection* and *Maximum-speed*. For DSVMP , We used a configuration of 5 VMs, DSVMP -5VM, in this experiment.

Code size. Figure 13 shows the impact on code size of several VM-based protection systems. From the figure we can see that the effects of DSVMP and commercial VM-based protection systems on code size are similar. Overall, the expansion of the code size is mainly affected by the size of the protected instructions, the more the target code brings more spatial overhead. Among them, the `aescrypt` has a greater code bloat after protected by VMProtect Maximum-protection, it may be caused by the complexity of the function `encrypt-stream()` and the Maximum-protection strategy.

Runtime overhead. The average runtime overhead of the three schemes are depicted in figure 14, which shows that the runtime overhead of DSVMP and VMProtect Maximum-speed are comparable and smaller than Code Virtualizer and VMProtect Maimum-protection. Code protected under CV and VMProtect Maimum-protection has the most expensive runtime overhead, which on average is higher than DSVMP and VMProtect Maimum-speed.

Discussion. Through the above analysis and comparison, we argue that the temporal and spatial overhead of our DSVMP are acceptable.

10. 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 [22], packers [23, 24], are used to protect software against attacks based on disassembly and static analysis. There are also other code protection techniques like code obfuscation [25], control flow and data flow obfuscation [26, 27, 28], 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.* [29] 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.

In recent years there are some deobfuscation methods have been proposed. Coogan *et al.* [30] puts forward a behavior based analysis method to analyze the important behavior of code, but it does not pay attention to how to restore the original code structure. Sharif *et*

al. [31] used dynamic data-flow and taint analysis to identify data and extract the syntactic and semantic information about the bytecode instructions. Yadegari *et al.* [32], by tracking the flow of inputs values, and then use semantics-preserving code transformations to simplify the logic of the instructions. These approach, however, can not restore the structure of the original code completely, and needed taint analysis to get the data flow.

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 [33].

11. 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, No. 61672427), 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, Y. Huang, Multi-stage binary code obfuscation using improved virtual machine., in: Information Security, International Conference, ISC 2011, Springer, 2011, pp. 168–181.
[5] M. Yang, L. S. Huang, Software protection scheme via nested virtual machine, Journal of Chinese Computer Systems 32 (2) (2011) 237–241.
[6] H. Wang, D. Fang, G. Li, N. An, X. Chen, 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, pp. 1–9.
[7] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, Y. Gu, Nislvmp: Improved virtual machine-based software protection, in: 9th International Conference on Computational Intelligence & Security (CIS), 2013, pp. 479 – 483.
[8] N. Falliere, P. Fitzgerald, E. Chien, Inside the jaws of trojan, Tech. rep., Clampi. Technical report, Symantec Corp (2009).
[9] C. Collberg, The case for dynamic digital asset protection techniques, Department of Computer Science, University of Arizona (2011) 1–5.
[10] R. Rolles, Unpacking virtualization obfuscators, in: 3rd USENIX Workshop on Offensive Technologies.(WOOT), 2009.
[11] C. S. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation-tools for software protection, IEEE Transactions on Software Engineering 28 (8) (2002) 735–746.
[12] Ida pro, https://www.hex-rays.com/index.shtml.
[13] Ollydbg, http://www.ollydbg.de/.
[14] Sysinternals suite, https://technet.microsoft.com/en-us/sysinternals/bb842062.
[15] B. Blietz, A. Tyagi, Software tamper resistance through dynamic program monitoring, in: Digital Rights Management. Technologies, Issues, Challenges and Systems, 2006, pp. 146–163.
[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] Pin - a dynamic binary instrumentation tool., https://software.intel.com/en-us/articles/pintool.
[21] Peering inside the PE: A tour of the Win32 portable executable file format., https://msdn.microsoft.com/en-us/magazine/ms809762.aspx.
[22] C. Linn, 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, pp. 290–299.
[23] Execryptor, http://www.strongbit.com/execryptor.asp.
[24] Upx, http://upx.sourceforge.net/.
[25] Z. Wu, S. Gianvecchio, M. Xie, H. Wang, Mimimorphism: a new approach to binary code obfuscation., in: ACM Conference on Computer and Communications Security (CCS), 2010, pp. 536–546.
[26] C. Liem, Y. X. Gu, 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, pp. 33–44.
[27] J. Ge, S. Chaudhuri, A. Tyagi, Control flow based obfuscation, in: Proceedings of the 5th ACM workshop on Digital rights management, 2005, pp. 83–92.
[28] V. Balachandran, N. W. Keong, S. Emmanuel, Function level control flow obfuscation for software security, in: Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), 2014, pp. 133–140.
[29] A. Averbuch, M. Kiperberg, N. J. Zaidenberg, An efficient vm-based software protection, in: 5th International Conference on Network and System Security (NSS), 2011, pp. 121–128.
[30] K. Coogan, G. Lu, S. Debray, Deobfuscation of virtualization-obfuscated software: a semantics-based approach, in: Proceedings of the 18th ACM conference on Computer and Communications Security (CCS), ACM, 2011, pp. 275–284.

[31] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: Security and Privacy (S&P), 2009 30th IEEE Symposium on, IEEE, 2009, pp. 94–109.
[32] B. Yadegari, B. Johannesmeyer, B. Whitely, S. Debray, A generic approach to automatic deobfuscation of executable code (2015) 674–691.
[33] P. Larsen, A. Homescu, S. Brunthaler, M. Franz, Sok: Automated software diversity, in: IEEE Symposium on Security and Privacy (S&P), 2014, pp. 276–291.