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

Kaiyuan Kuang[a], Zhanyong Tang[a,], Xiaoqing Gong[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 builds 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 always follows a single, deterministic execution path for the same input. Such approaches, however, are vulnerable in certain scenarios where the attacker can reuse knowledge extracted from previously seen software to crack applications protected with the same obfuscation scheme. 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 approaches. 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 path, 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 launch an attack. Secondly, it employs multiple VMs to further obfuscate the mapping from VM opcode to native machine instructions, so that the same opcode could be mapped to different native instructions at runtime, 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 it is compared to two commercial VM-based code obfuscation tools.

Keywords: Code virtualization, Code Obfuscation, Reverse Engineering

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

1. Introduction

Unauthorized code analysis and modification based on reverse engineering is a major concern for the software industry. Such attacks can lead to a number of undesired outcomes, including cheating in online games, unauthorized use 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 [2, 3, 4, 5, 6, 7, 8]. The underlying principal of VM-based protection is to replace the program instructions with virtual instructions which attackers are unfamiliar with. These virtual instructions will then be translated into native machine code at runtime to be executed 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 instruction is ready for execution, and a set of *bytecode handlers* that first decode the bytecode[2] and then translate it into native machine code. This process replaces the original program instructions with bespoke bytecode, 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 single virtual instruction scheduler. This is based 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 this is an unreliable assumption [9] in certain scenarios where an adversary can easily reuse knowledge obtained from other applications protected with the same scheme to preform reverse engineering (termed *cumulative attacks* in this work). To protect software against cumulative attacks, it is important to have a certain degree of non-determinism and diversity during program execution [10].

This paper presents DSVMP (*dynamic scheduling for VM-based code protection*), a novel VM-based code protection scheme to address the problem of cumulative attacks. Our key insight is that it will be more difficult for the attacker to track the application logic if sensitive code regions behave differently in different runs. DSVMP achieves this by introducing rich non-determinism and diversity to 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 for each

Email addresses: kky@stumail.nwu.edu.cn (Kaiyuan Kuang), zytang@nwu.edu.cn (Zhanyong Tang), gxq@nwu.edu.cn (Xiaoqing Gong), dyf@nwu.edu.cn (Dingyi Fang), xjchen@nwu.edu.cn (Xiaojiang Chen), z.wang@lancaster.ac.uk (Zheng Wang)

[2]A bytecode is the binary form of a virtual instruction.

protected code region. As a result, the program follows different execution paths in different runs and exposes an non-deterministic behavior. Our carefully designed scheme ensures that the program will produces a consistent output for the same input despite the execution paths might 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 the logic of the program. This significantly increases the cost of code reverse-engineering.

Dynamic instruction scheduling in DSVMP is achieved through a combination of two techniques. Firstly, DSVMP provides a rich set of bytecode handlers, each of which has a unique control flow, to translate a bytecode instruction to native code. Handlers for a particular bytecode opcode all generate identical native machine instructions 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 a bytecode handler to translate a bytecode to the corresponding 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 across different executions. 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 the diversity of the program, making it even harder for an adversary to analyze the software behavior or to reuse knowledge extracted from other software products. This is because different products are likely to be protected using different bytecode forms 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 applications that implement some of the widely used algorithms: "`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 [3] and VMProtect [4].

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.

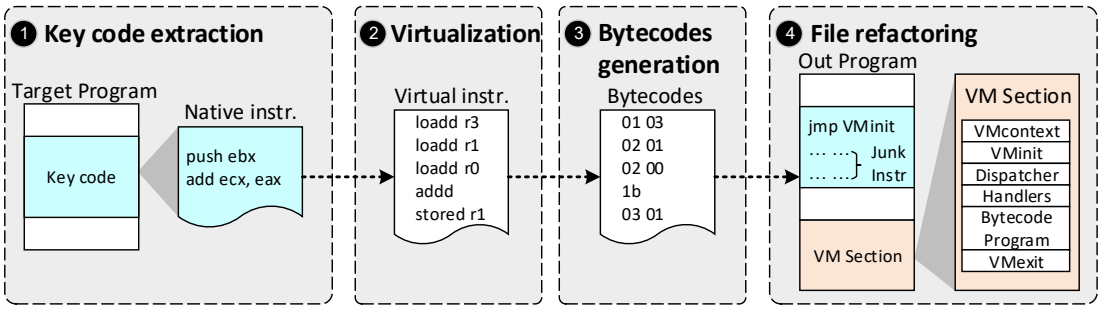


Figure 1: A classical process for VM-based code obfuscation. To obfuscate the code, we first dissemble the code region to be protected into native assembly code (1). The assembly code will be mapped into our virtual instructions (2) which will then be encoded into a bytecode format (3). Finally, the generated byecode will be inserted into a specific region of the binary which is linked with a VM library (4).

2. Background

2.1. VM-based Code Obfuscation

VM-based code obfuscation often performs at the binary level for an already compiled program. As shown in Figure 1, the obfuscation process typically follows a number of steps. Firstly, the critical code segment to be protected will be extracted from the compiled binary, which will be dissembled into assembly code. Next, the native assembly code will translated into virtual instructions, i.e. a machine-independent intermediate representation used by our VMs. The translated virtual instructions are functional equivalent to original native code. Then, the generated virtual instructions will be encoded into the bespoke bytecode format. Finally, a new VM section will be linked (or inserted) into the target program where the entry point of the protected code region will be redirected to a function call to invoke the VM to translate the bytecode instructions to native machine code at runtime.

The idea of VM-based code obfuscation is to force the attacker to move from a familiar instruction set (e.g. x86) to an unfamiliar bespoke virtual instruction set, which hopefully will significantly increase the time and efforts for reverse-engineering.

2.2. VM Components.

Our approach follows a classic VM implementation, consisting of a number of components that are shown in step 4 at Figure 1. The context of the native program, which includes information such as local variables, function arguments, the return address etc., will be stored in a register-based VM memory space called `VMContext`. When entering the VM, the `VMinit` component saves the native program context and initializes the `VMContext`. After executing the protected code segment, `VMExit` restores the native program context, and then returns the program control back to the original program to continue executing native machine code.

At the heart of the VM is an interpreter consisting of a dispatcher and a handler set described as follows. The dispatcher fetches a bytecode that is ready to be exeucted, decoding the fetched bytecode (by parsing the opcode and the operand), and then assigning a handler

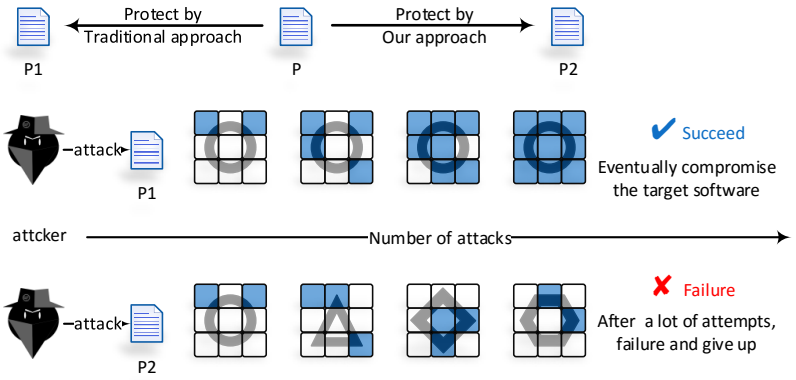


Figure 2: Diversity affects the attack effectiveness. In this example, a dark small square represents reusable attacking knowledge. Diverse program execution increases the difficulty for performing reverse-engineering based attacks.

(from a collection of handlers that can be used to interpret the bytecode) to translate the fetched bytecode to native machine code. This process iterates until all the bytecode of the target protected code region are executed. For the attacker's perspective, the key to understand the logic of the protected code region is to find out how bytecode or virtual instructions are mapped into native machine code.

2.3. The Design Goal

Figure 2 gives a high-level abstraction on how an attacker can reuse knowledge extracted from the previous runs of the same application or other applications protected under the same VM scheme to perform reverse-engineering. This kind of attacks is referred as *cumulative attacks* in this paper.

In the first scenario, the software always follows the same execution path across multiple runs. Under this setting, the attacker may be able to obtain sufficient knowledge on the program behavior in a few trail runs. 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 is key for us to protect software against dynamic cumulative attacks. This is the aim of this work, to improve the diversity of program executions for code obfuscation. It is to note that like any other code protection techniques, our approach could also be exploited by malware. How to prevent this is out of the scope of this work.

3. The Attack Model

3.1. Attacking methods

The classical approach to reverse engineer a VM-protected program typically follows three steps [9, 11], described as follows. The first step is to understand how each components of a VM interpreter works. To do so, the attacker needs to locate these components and analyze how the dispatcher schedules bytecode instructions for interpretation. The second step is to understand how each bytecode is mapped to machine code and work out the

semantics of the bytecode instructions, i.e. how will a bytecode opcode be translated into a native machine instruction. The third step is to use knowledge obtained in the first two steps to recover the logic of the target code region, through e.g. removing the redundant information and generating a simplified program that is equivalent to the original program.

To perform such an attack, a significant portion of the time will have to spend in analyzing the working mechanism of the VM. The problem is that a skilled attacker could reuse knowledge gathered from parts of the program to analyze other protected code regions of the same program, or other applications protected using the same VM scheme and bytecode instructions.

3.2. The Threat model

Our attack model assumes that the attacker has the necessary tools and skills to implement the above reverse-engineering based attacks. We assume that the adversary holds an executable binary of the target software and can run the program in a control environment [12]. We also assume that the adversary can access content stored in memory and registers, trace and modify program instructions and control flows. All these can be achieved using sophisticate profiling and analysis tools like "IDA" [13], "Ollydbg" [14] and "Sysinternals suite" [15]. 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 protected using VM-based code obfuscation.

4. Overview of Our Approach

To address the problem of cumulative attacks, we want to introduce a certain degree of diversity and non-determinism to the program execution. This is achieved through using a diversified scheduling structure (Section 5) and multiple VMs (Section 6). Like any VM-based protection schemes, DSVMP should be used to protect the most critical code regions but not the entire program, in order to minimize the runtime overhead. Our current implementation targets the Intel x86 instruction set but the methodology itself can be applied to other instruction sets and hardware architectures. Figure 3 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. It does not require having access to the source code. Code segments need to be protected are given by providing the symbolic name of the target functions or the start and end addresses of a code block. The code segments are firstly converted into native machine assembly code (e.g. x86 instructions) using a disassembler (Step ❶). The assembly code will then be mapped into a set of virtual instructions, i.e. the intermediate language used by the VM (Step ❷). The virtual instructions will then be stored in a bytecode format.

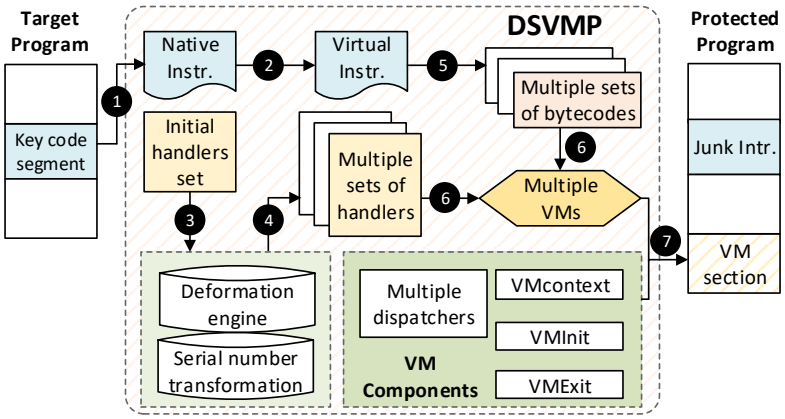


Figure 3: Offline code protection process. DSVMP takes in a program binary. For each protected code region, it translates native instructions into bytecodes. Next, it generates multiple bytecode handlers that are semantically equivalent but implemented in different ways. It then generates the corresponding 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. Each virtual instruction scheduler can have multiple dispatchers and bytecode handlers. DSVMP provides a set of handlers that are semantically equivalent but are implemented in different ways for a single virtual instruction. Thus, the scheduler can dynamically determine at runtime which of the handlers is used to decode a bytecode (i.e. the encoding scheme of the virtual instruction which includes the opcode and operand) and to interpret a virtual instruction. Multiple handlers are generated by applying obfuscation to a set of seed handlers (Step ❸). The way the handlers are obfuscated could be different for different code regions. DSVMP also employs a multi-VM scheme by providing more than one VM implementation. Therefore, each handler will be obfuscated for each VM by using the deformation engine (i.e. a obfuscation toolkit), resulting in n (i.e. the number of VMs) sets of semantically equivalent handlers with different implementations and control flows (Step ❹). Next, the virtual instructions will be encoded into an unique bytecode form for each VM, so that the same opcode from different VMs will be mapped into different native machine instructions to protect against static analysis. Our preliminary implementation provides two sets of bytecode in a VM. Therefore, each virtual instruction can be encoded into two different sets of bytecode forms in a VM(Step ❺). After these steps, DSVMP essentially provides multiple VMs, where each VM contains one set of bytecode handlers (so that a virtual instruction can be interpreted by multiple handlers), while the instructions of the protected code regions are stored in different bytecode forms (Step ❻).

Code generation. Finally, a new section will be inserted into the program binary, which contains n VMs and their components such as dispatchers, `VMContext` etc. Because the size of the generated VM code and virtual instructions is typically larger than the protected code region, and a PE file on disk, each section follow a certain file alignment value (512

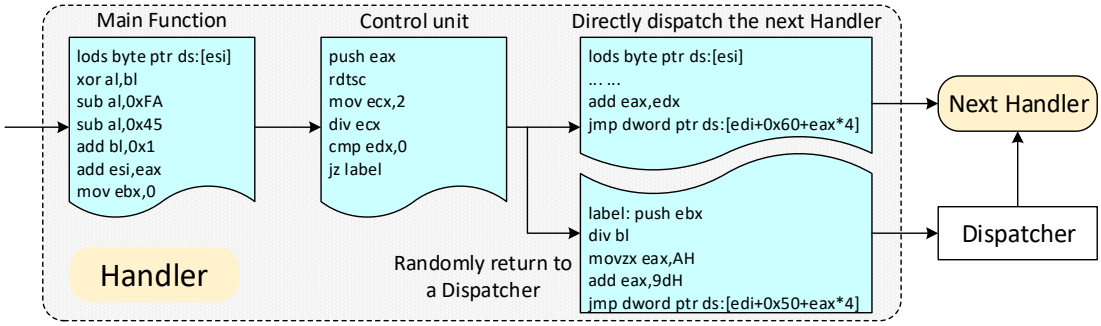


Figure 4: The execution flow of the new handler, each bytecode handler has a control unit that randomly determines whether the control after exiting the handler should be given to a dispatcher or an alternative bytecode handler.

byte) [16], we simply fills the original code region with junk instructions(Step ❼), rather than utilizing the space to store the VM code.

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 is dynamically changed at execution time. To further increase the diversity of the program's behaviour, Dsvmp also uses multiple bytecode instruction sets and bytecode handlers.

5.1. Multiple Bytecode Handlers

In classical VM-based code obfuscation, a single dispatcher is responsible for fetching a bytecode instruction, and then determining which bytecode handler to use by examining the opcode of the bytecode instruction. Because each bytecode instruction is decoded by a fixed handler set, an adversary can easily work out the mapping from an opcode to its handler. From the mapping, the adversary can correlate the native machine code to each bytecode to analyze the program behavior and the logic structure.

To address this issue, for each bytecode handler, we use obfuscation techniques to automatically generate a number of alternative implementations which all produce an equivalent output for the same input instruction. Different implementations, however, are programmed in different ways using e.g. different control flows, data structures or obfuscation methods.

To control the program execution path, 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 back to a dispatcher or another handler. Figure 4 shows an example of the control unit of a Dsvmp bytecode handler. When the main function of the handler is finished, the control unit randomly switches to one of the branchs. At the first branch, the "`lods`" (a load operand in the x86 assembly) instruction first fetches an offset value from *Offset Bytecode* to calculate the address of the nest bytecode handler, and

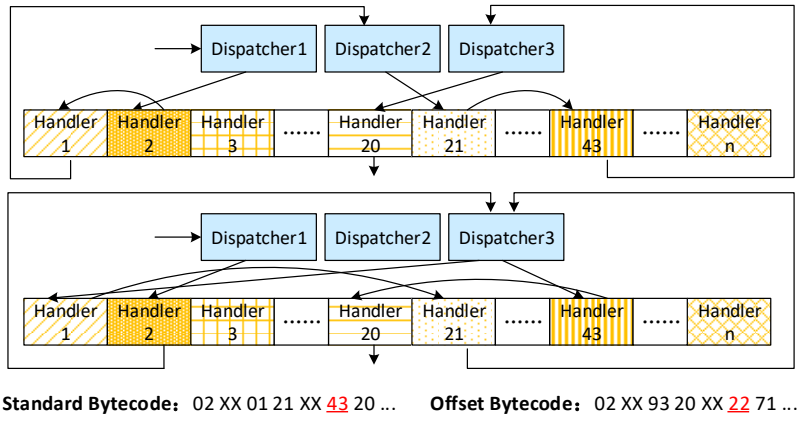


Figure 5: Our approach employes multiple dispatchers together with a control unit to schedule the handlers. In this example, the type of handlers and the order for invoking the handlers could be different across execution runs. The "XX" in bytecode refers to the parameters. Underlined two bytecodes is handler43 encoding results in two types of *Bytecode* respectively

then jump to execute it. By contrast, the instruction at another branch will return to a dispatcher. Figure 5 shows two different dynamic scheduling results. Under the instruction of the control unit, a handler can either be invoked by a dispatcher or a handler. The *Offset Bytecode* here is different from the usual standard bytecode, which is designed to drive the handler's dispatch function, and we will describe it in detail in the next section.

5.2. Multiple Bytecode Formats and Dispatchers

Bytecode Formats. Using VM-based obfuscation, native machine code of the protected code regions will be translated into virtual instructions and stored in a bytcode format. A handler will be chosen to decode the bytecode instruction to translate it back to native machine code at runtime. In classical VM-based code obfuscation approaches, there is one-one mapping from a bytecode opcode to a handler, i.e. the bytecode opcode determines which handler to be used. Having multiple bytecode instruction sets for different code regions of a target program can provide stronger protection. By doing so, the same opcode from different code region will have different semantic meanings. This is because the same opcode from different regions can be mapped to different handlers (and hence different native machine code implementations). For this reason, the virtual instructions of different code regions will be stored in different bytecode formats.

Our current implementation uses two bytecode formats for each VM, namely *Standard Bytecode* and *Offset Bytecode*. Additional bytecode formats can be easily added to our system. Here, *Standard Bytecode* is a standard bytecode format where each bytecode consists of the virtual instruction's opcode[3] and their operands. Instructions encoded as *Standard Bytecode* will be fetched and executed by the dispatcher. *Offset Bytecode* uses a different encoding scheme, which is fetched by the handler's control unit. Each *Offset Bytecode*

[3]The opcode is a ID indicates which handler should be used to interpret the virtual instruction.

consists of the offset IDs of two handlers (e.g. handler21 and handler43 in Figure 5 has an offset ID of 22) and their operands respectively. Recall that a control unit is inserted to the end of each handler to determine whether the control should be given back to the dispatcher or another handler. Before exiting the current handler, if the control unit chooses to execute the next handler, it will fetch the corresponding offset ID from *Offset Bytecode*. The offset ID will then be used to calculate the id of the next handler to execute.

Dispatchers. DSVMP also provides multiple dispatchers to further increase the diversity of program execution. As an example, considering Figure 5 which shows two possible program execution paths using three dispatchers within a single VM. As can be seen from the diagram, each time when a different dispatcher is chosen, a handler can either be invoked by a dispatcher or another handler; and the type of handlers to be invoked could be different in two different execution runs. As a result, 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 (*termed 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 possible mapping from a bytecode instruction to a handler. Together with the multiple scheduling approach described above, DSVMP can further increase the diversity and uncertainly of program execution.

6.1. Switching between Multiple VMs

The number of VMs to use is a parameter provided by DSVMP. This can be configured by the user. This number can vary depending on the target program to be protected, and the trade-off between the protection strength and runtime overhead. As described in Section 5.2, we generate a set of handlers for each VM so that we have n different sets of handlers for n VMs. Our current implementation also translates the virtual instructions of each handler set to stored as two different sets of bytecode. Different bytecodes will have different semantics in different VMs. Therefore, there are more than a bytecode that can be translated by different handlers in different VMs.

Our system dynamically determines which VM to use at runtime. This is done through altering the structure of the instruction dispatcher that decides which VM to use at runtime. Figure 6 shows an example of the new dispatcher structure, which has a VM switch unit that can randomly select one of the multiple VMs to use. To do so, The switching unit first randomly selects one VM among the multiple VMs (lines 1-7), and then modifies the virtual program counter (`VM PC`) to point to the target VM (lines 8-10). Our implementation uses the x86 `ESI` register as a `VM PC` to store the address of the next bytecode instruction. Other registers can also be used for this purpose. Finally, the dispatcher fetches a bytecode according to the modified `VM PC` and dispatches the handler to interpret it in the corresponding

```
1  push edx                                ;-------------------------
2  rdtsc
3  xor edx,edx
4  div dword ptr ds:[edi+0x58]             ;VM switching unit
5  mov eax,edx
6  sub edx,dword ptr ds:[edi+0x50]
7  je label                                ;-------------------------
8  imul edx,dword ptr ds:[edi+0x54]        ;Modify the VM PC and
9  add esi,edx                             ;save the current VM ID
10 mov dword ptr ds:[edi+0x50],eax
11 label: lods byte ptr ds:[esi]           ;-------------------------
12 ... ...                                 ;Fetch the bytecode and
13 movzx eax,al                            ;dispatch a Handler
14 add eax,edx                             ;to interpret it
15 pop edx
16 jmp dword ptr ds:[edi+eax*4+0x60]
```

Figure 6: The new dispatcher has a VM switch unit that can randomly select one of the multiple VMs to continue scheduling handler.

VM (lines 11-16). 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. The VM Scheduling Process

Our dynamic scheduling is achieved through two control units: (1) a structure control unit to randomly determine whether the execution control should be given to a dispatcher or another bytecode handler, (2) a VM switching unit to randomly select a VM to use. Having these two control units to switch the execution path of virtual instruction interpretation can greatly increase the diversity of the program behavior when compared to existing approaches that have a single, fixed scheduling structure.

Our dynamic scheduling scheme is described in Algorithm 1. Bytecodes of a code region will be executed one after one in sequential order. The virtual interpreter fetchs a bytecode from *Standard Bytecode* and dispatchs a handler to interpret the bytecode (lines 5-6). After executing the bytecode, the control unit will randomly decide whether the control should be given back to a bytecode dispatcher or a VM handler (line 7). If the control is given back to a bytecode dispatcher (lines 8-11), a dispatcher and a VM will be randomly chosen to execute a bytecode from *Standard Bytecode*. If the control is directed to another bytecode handler (lines 12-13), the program will execute the next bytecode from *Offset Bytecode* . The process continues until all the virtual instructions of the protected code region have been executed.

7. An Example

We use a short x86 code snippet shown in Figure 7 as an example to illustrate how DSVMP operates. In this example, "`STARTSDK`" and "`ENDSDK`" are used to mark the begin

Algorithm 1 Virtual Interpreter's Work Flow

```
VMInit
Switcher selects a VM randomly
Fetch a bytecode from Standard Bytecode in current VM
while bytecode ≠ ∅ do
    Decode the bytecode
    Select a handler to interpreter the bytecode
    i = handler_exit_address
    if i == dispatcher then
        Select a dispatcher randomly
        Switching unit selects a VM randomly
        Fetch the next bytecode from Standard Bytecode from the selected VM
    else {i == handler}
        Fetch the next bytecode from Offset Bytecode in the current VM
    end if
end while
VMExit
```

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

Figure 7: Example assembly code snippet for a code region to be protected.

and end of the code region respectively, and "00401036" and "00401038" are the address of two assembly instructions.

7.1. Code obfuscation

Table 1 shows the resulted obfuscated code for the code example given in Figure 7. Firstly, DSVMP extracts the target code region and disassemble it into native instructions. DSVMP inserts two additional instructions ("`push 0x40103b`" and "`ret`") at the end of the protected code region, in order to jump back to execute the native code. It then converts the native instructions to virtual instructions based on a translation convention. 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, our implementation provides two VM 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: *Standard Bytecode*

Table 1: Generated virtual instructions for the example shown in Figure 7

	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.

(Set11) and *Offset Bytecode* (Set12) for `HSA1` and *Standard Bytecode* (Set21) and *Offset Bytecode* (Set22) for `HSA2`. The resulted program is illustrated in Figure 8. We store the virtual instructions as bytecode. Because the resulted code size is larger than the original code and hence cannot simply be used to replace the original instructions. For this reason, we simply fill the original code region with junk instructions.

Finally, DSVMP creates 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 cleaning up the context before exiting the VM).

7.2. Runtime execution

Runtime execution of the protected code region is illustrated in Figure 8, which follows a number of steps:

- **Step 1:** 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, `VMContext`.
- **Step 2:** Next, a dispatcher is used to schedule the virtual instructions. It randomly selects a VM (the example shown in Figure 8 assumes that `VM2` is chosen at beginning) and then it fetches a bytecode from the *Standard Bytecode*, Set21. After decoding the bytcoe, the dispatcher gets an operand, "`6a`", which directs the dispatch to jump to execute another handler, "`0x6aHandler`".
- **Step 3:** A control unit is executed (see Section 5.1) before exiting the "`0x6aHandler`" handler. The control unit determines at runtime whether to execute another handler or to return the control to the dispatcher. If it chooses to return the control to the dispatcher, the program execution moves to Step 5.
- **Step 4:** Assume that the control unit decides to execute the next handler. It will fetch a bytecode from *Offset Bytecode*, Set22. Adding the offset of 22 to the current handler "`0x6aHandler`" gives the address (`0x85`) of the next handler "`0x85Handler`". The

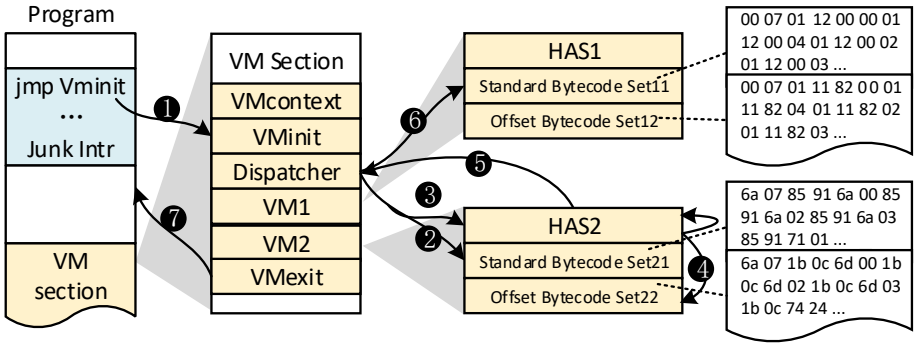


Figure 8: The execution process of the protected program. Here each VM has two sets of bytecode instructions and one set of handlers.

control unit will then jump to execute "`0x85Handler`". After executing this handler, the program execution loops back to Step 3.

- **Step 5:** If the control unit chooses to return the control to a dispatcher, it will randomly select a dispatcher to continue the execution. Then, the program moves to Step 6.
- **Step 6:** The selected dispatcher randomly selects one of the VMs to use. The dispatcher fetches a bytecode from the selected VM, decoding it to get the handler's address. It then jumps to execute the handler. After executing the handler, the program goes back to Step 3.
- **Step 7:** Steps 3 and 4 are iterated until all the virtual instructions get executed. The finally step is to invoke the `VMExit` procedure to restore the native execution context to continue executing the rest native code of the program.

8. Security Strength Analysis

In this section, we evaluate the security strength of Dsvmp. We first analyze the number of possible execution paths, showing that Dsvmp can significantly increase the diversity and non-determinism of program execution. Then, we discuss how Dsvmp can enhance 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 runs the protected region does not follow a single execution path across runs. To make the analysis concrete, we assume that there are 10 different dispatchers, which is the standard setting in the current implementation of Dsvmp. We use the example presented in Section 7 as a case study. In this example, *Standard Bytecode* 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 and will always get executed. 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 small possibility. It is to note that so far we have assumed that the protection scheme uses just one VM. The multi-VM strategy employed by DSVMP can further increase the number of possible execution paths. This is because the more dispatchers and VMs are, the greater number of possible execution paths will be. For example, if the DSVMP implementation provides five different VMs, then each dispatcher can randomly select one VM among the 5 VMs; as a result, each handler can lead to 51 $(= 10 * 5 + 1)$ different execution paths. Together with multiple dispatchers and bytecode instruction strategies, for the setting used in this discussion, DSVMP gives a single code region 51^{77} possible execution paths.

In summary, we can conclude that for a single code region protected by DSVMP, its possible execution path number, m, and the probability of its execution of the same execution path across different runs, p, respectively are:

$$m = (N_D * N_{VM} + 1)^n, \quad p = 1/m.$$

where N_D represents the number of dispatcher, N_{VM} is the number of VMs, and n is the number of handler scheduling options. Given the massive number of choices, it will be unlikely for a protected code region to follow the same execution path across different runs.

8.2. Code structures

Having a diverse code structure is key to prevent an adversary from reusing knowledge obtained from other software to launch a new attack. In other words, we would like the code structures of the obfuscated program's to be as dissimilar from the original program's as possible.

Blietz *et al.* [17] proposed a method to measure the similarity of program structures. Their method is based on the control flow information such as the number of branches and back blocks, the nesting level of the code etc. We adopt the metrics from [17] to analyze code structures for programs protected using DSVMP. We use a number of metrics to quantity the code structure. 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.

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`.

- $DF(Vi)$: 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.

8.2.1. Example

Table 2 illustrates two code segments to be protected. These two code snippets have very similar structures because they both have one basic block and there is no branch within the basic block. As can be seen from the table, the code transformation applied by DSVMP leads to significant variances in the metric values. This indicates that the transformed code segments have distinct code structures. The metric value calculation is described as follows.

We use the following formula to quantify the code structure information, $SInfor_X$, for a given piece of code, X, after code obfuscation.

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

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

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

From $SInfor_{A'}$ and $SInfor_{B'}$, we can calculate the similarity $SDiff$, for the two piece of code, A' and B', as:

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

Table 3: Information of the benchmarks

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
astar	281	29036	`SearchPath()`	156	19589
tetris	37	4386	`ClearRow()`	73	10942

Notes: The 3rd column shown the total number of target program instructions. The 4th 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.

Therefore, the quantified code structure similarity between A' and B' is 34.5%. This example shows that DSVMP can significantly increase the dissimilarity of code structures even for simple code segments.

9. Performance Evaluation

We now evaluate the performance of DSVMP using six applications and compare it with two commercial VM-based protection systems.

9.1. Evaluation Platform and Benchmarks

We evaluated DSVMP on a PC with an 3.0 GHz Intel Core-i3 Duo processor and 8GB of RAM. The PC runs the Windows 10 operating system.

We evaluated our approach using four widely use applications, `md5`, `aescrypt`, `bcrypt`, `gzip`. We used these applications to process a test text file. The size of the file is 26 KB. In addition to the four applications, we also evaluate our approach on two interactive gaming applications: "`AStar`", a maze pathfinding game using the A* algorithm, and"`Tetris`", a classic graphics combination game [18]. Figure 9 shows the "`AStar`" maze barrier setting and route finding results, and the clear row operations of "`Tetris`". 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 in Table 3. The 4th 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 [19] to calculate the number of instructions get executed within the protected functions while processing the test file. The result is shown in the last column of the table.

9.2. Code Size and Runtime Overhead

9.2.1. Code size

For each target benchmark, we will choose a core function to protect. We do so by inserting a set of specific SDK ("STARTSDK" and "ENDSDK") at the beginning and end of the target function in the compiled program binary. We applied DSVMP to the target function and repeated the process for 10 times. For each protection run, we used a different number of VM configurations.

Figure 10 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; therefore, it is

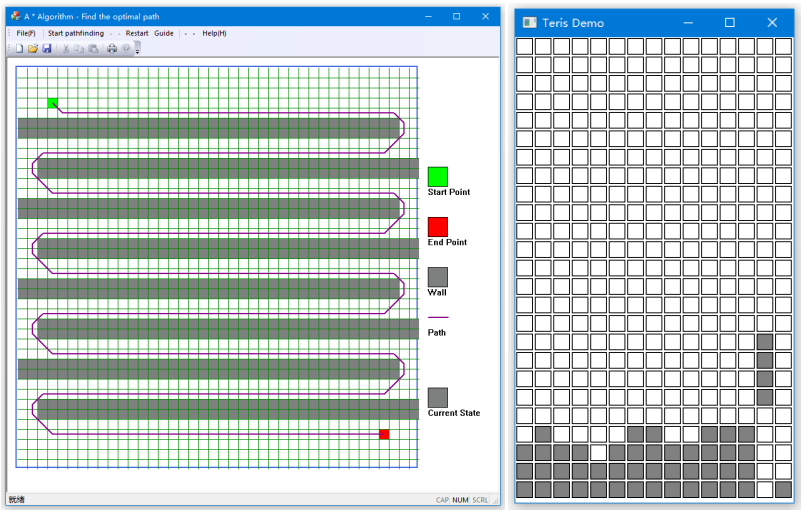


Figure 9: The interaction interface of the "`AStar`" and "`Tetris`".

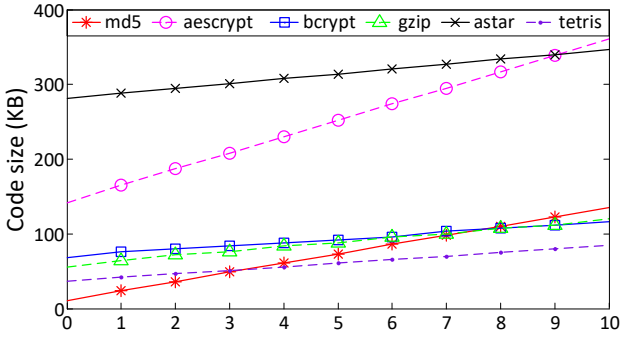


Figure 10: The impact of code sizes for DSVMP configurations with a different number of VMs. The number of horizontal axis is the configuration of the VMs, "0" is the original program.

not supervising that the code size of the protected program grows as the number of VM increases. In general, the code size grows as the number of VMs increases, because the block of PE executables are usually follow a certain alignment value (such as, 4096 or 512 byte) [16]. Moreover, we can see that there is a strong correlation between the code size and the number of protected instructions. This is why the code size of "`aescrypt`" grows fastest than others – 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. Overall, the code size growth (a few hundreds KB in our experiments) is modest as typically we only need to protect the core function or a core algorithm of an application.

9.2.2. 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. For "`AStar`" and "`Tetris`", in order to eliminate the user

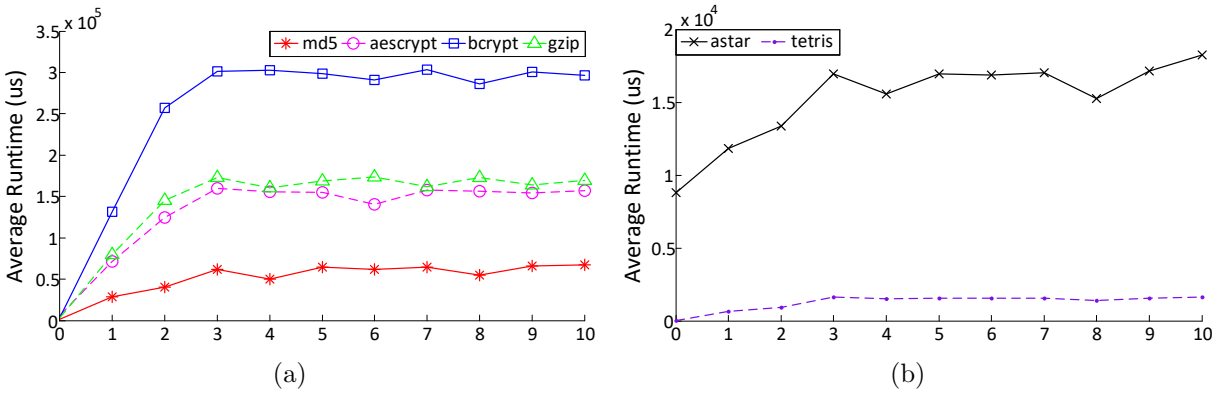


Figure 11: The average runtime of target benchmark when protected with different VMs. The number of horizontal axis is the configuration of the VMs, "0" is the original program. Because the runtime overhead of "`AStar`" and "`Tetris`" is much smaller than the other four benchmarks, in particular, their experimental data are shown separately in (b) in order to observe their variation.

Table 4: Possible execution path statistics

Basic configuration			m, **for different** N_{VM} **configurations**							
program	N_D	n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
md5	5	6175	6^{6175}	11^{6175}	16^{6175}	21^{6175}	26^{6175}	31^{6175}	36^{6175}	41^{6175}
aescrypt	5	8014	6^{8014}	11^{8014}	16^{8014}	21^{8014}	26^{8014}	31^{8014}	36^{8014}	41^{8014}
bcrypt	5	585	6^{585}	11^{585}	16^{585}	21^{585}	26^{585}	31^{585}	36^{585}	41^{585}
gzip	5	1271	6^{1271}	11^{1271}	16^{1271}	21^{1271}	26^{1271}	31^{1271}	36^{1271}	41^{1271}
astar	5	1502	6^{1502}	11^{1502}	16^{1502}	21^{1502}	26^{1502}	31^{1502}	36^{1502}	41^{1502}
tetris	5	696	6^{696}	11^{696}	16^{696}	21^{696}	26^{696}	31^{696}	36^{696}	41^{696}

Notes: The specific calculation process is described in section 8. Where N_D is the number of dispatcher, N_{VM} is the number of VMs, and n is the number of handler scheduling.

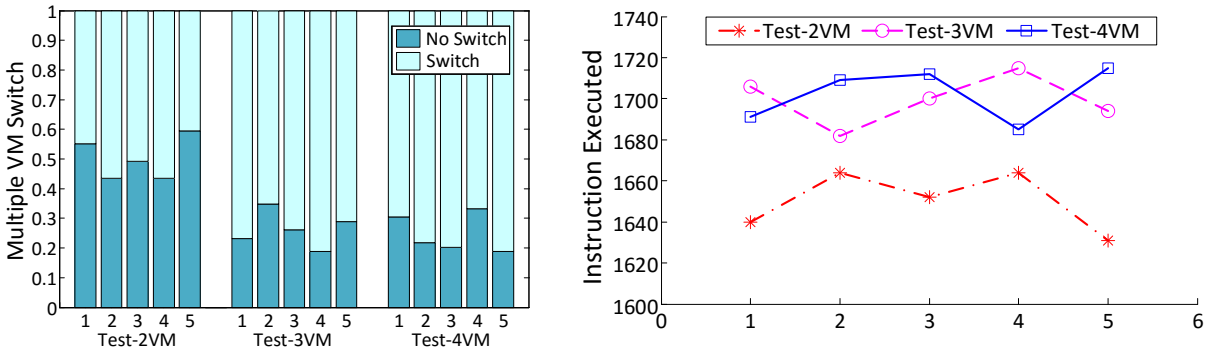
interaction delay, we only calculate and collect the average runtime of the target operations (pathfinding and row clear).

The results are depicted in Figure 11. We see an increase on the runtime overhead when using multiple VMs but the overhead becomes stable from 3 VMs onward. which shows that the majority of average runtime. Except for the handler's normal interpretation of the time spent on the execution, the main runtime overhead comes from the the switch time between different VMs. The overhead of VM switch is stable so that it does not significantly increase when using 3 or more VMs. We notice that the overhead of VM switch for using two VMs is not significant. This is because our scheme does not frequently switch the VM under a 2-VM configuration. Besides, we found that the greater the number of protected instructions get executed at runtime the greater the runtime overhead will be. This explains why "`bcrypt`" has a much higher runtime overhead than other benchmarks.

9.3. Structural diversity

9.3.1. Possible execution path statistics

In the previous section, we have discussed how to calculate the number of optional execution paths for a code region protected by DSVMP (see Section 8.1). We adjust the output of the protection system to collect the number of handlers that need to be scheduled



(a) (b)

Figure 12: (a) is the probability distribution of multiple VMs switching for multiple runs. The legend "`switch`" donates the number of times a VM has been switched to another, and the "no switch" means the VM remains unchanged. (b) is the number of instructions changes across runs due to the VM switch.

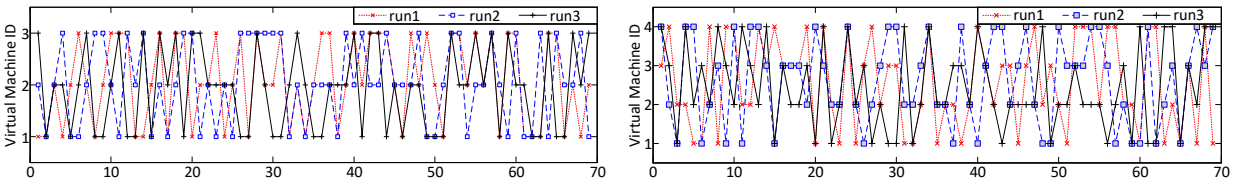
for different benchmarks. We then calculate the values of m under different configurations for these benchmarks, based on the formula presented at Section 8.1. Detailed experimental data are shown in Table 4. From the experimental data, we can clearly see that the number of optional paths for each benchmark is a huge number, and the program protected by DSVMP rarely follows the same execution path across different runs. At the same time, through the previous experimental data we found that the number of runtime instructions for the target code is much larger than the number of its original instructions (see Table 3). This is because the critical method is executed multiple times at runtime. Therefore, even in a single run for a program protected by DSVMP, calls to the critical function is likely to follow different execution paths during a single run.

9.3.2. Multi-VM switching experiment

In order to evaluate the impact of multi-VM switching on program execution, we conduct experiments on using a micro-benchmark, "`test.exe`[4]. Here we only protect one instruction "`mov eax, 1234567`" of the application, in order to isolating other scheduling effects. In this experiment, we do not perform handler obfuscation and used only one dispatcher in the protection process. This can minimize the impact of irrelevant factors. We use three VM configurations, 2-VM, 3-VM and 4-VM, to protect the target program. For each configuration, we run the program execution 5 times to collect the relevant information.

Figure 12 (a) shows the VM switching frequency information of "`test.exe`" for the five runs. We observe that the frequency of VM switching is low when 2 VMs are use. However, the frequency increases by 75% when 3 or 4 VMs are used. Through the experiment we found that the impact of 2-VM on the number of instructions gets executed at runtime is less than 3-VM and 4-VM. This is depicted in Figure 12 (b). Essentially, the number of instructions gets executed can reflect the runtime overhead. This explains why the 2-VM configuration has a smaller effect on runtime overhead in Figure 11.

[4]A 3KB application that pops up a confirmation box.



(a) (b)

Figure 13: Dynamic VM switching for executing protected code regions. There are a total of 69 handler schedules. The y-axis shows the VM selected in each schedule. Different lines show the VM switches across runs for a 3-VM (a) and a 4-VM (b) configuration.

9.3.3. 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 13 shows the switching of the VM for three 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 across different runs, and the execution path of the 3 runs is not the identical. For each dispatch loop, the dispatcher will randomly selects a VM to enter and schedules the corresponding handler. As mentioned before, the handlers set in each VM is obfuscated with different obfuscation schemes. Thus, the internal structure of the handler dispatched from different VMs during each run is different. The experimental results show that the execution path of a protected program exhibit strong diversity. Combined with various handler sets, the target program protected by DSVMP will have temporal diversity [10].

9.4. Comparisons with state-of-the-arts

We also compared DSVMP against two commercial VM protection systems, Code Virtualizer (CV) [3] and VMProtect (VMP) [4], in terms of code sizes and runtime overhead. We adopt a customized protection scheme when using CV to protect the target program. Specifically, this scheme uses the *Medium* opcodes obfuscation options, and does not use the "Strip Relocation", "Re-Virtualization" and other additional code obfuscation schemes. Doing so allow us to keep the runtime and code-size overhead of the CV scheme at a moderate level, in order to provide a fair comparison. For VMP, we use two types of schemes to protect the target program, *Maximum-protection* which provides the strongest protection (noted as VMProtect-Maximum-protection), and *Maximum-speed* (noted as VMProtect-Maximum-speed), which aims to reduce the overhead. For DSVMP, We use a configuration of 5 VMs, DSVMP-5VM, in this experiment. This is because the code-size overhead of 5-VM configuration is moderate.

9.4.1. Code size

Figure 14 shows the impact on code size of several VM-based protection systems. From the figure we can see that DSVMP has a similar code-size overhead, if not smaller, when compared with CV and VMP. For example, for the least affected program `bcrypt`, the code sizes of the target function is around 120% for all schemes. For `aescrypt`, the code-size overhead of DSVMP-5VM and CV is around 177%. This is significantly smaller than the

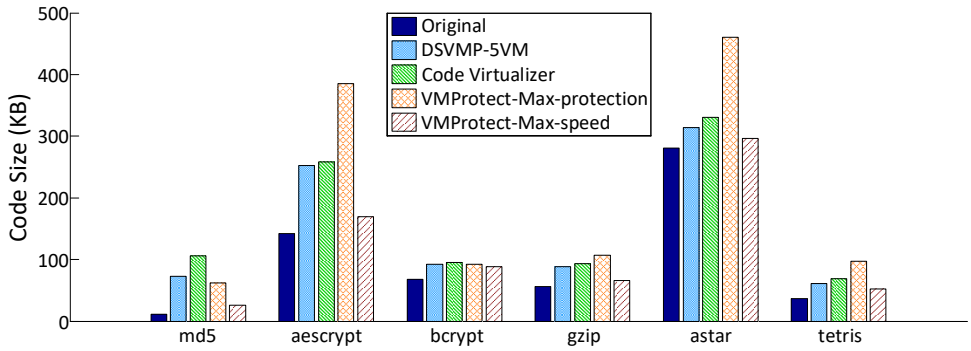


Figure 14: Code size comparisions.

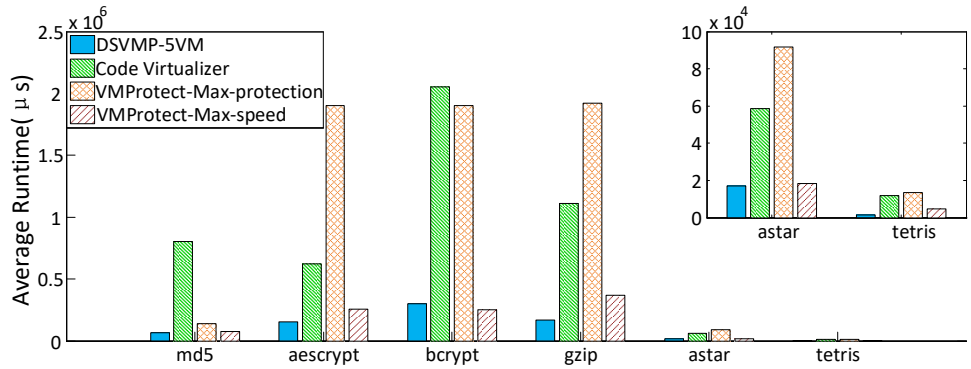


Figure 15: Average runtime overhead comparisions.

Table 5: The average runtime overhead(ms)

program	Original	DSVMP 5VM	VMProtect Max-Speed	VMProtect Max-Protect	Code Virtualizer
md5	1.603	64.229	73.660	141.179	803.302
aescrypt	6.550	154.720	257.718	1903.458	623.452
bcrypt	4.544	298.415	252.655	1899.968	2051.708
gzip	3.963	169.107	368.599	1923.022	1110.598
astar	8.806	16.992	18.210	91.691	58.705
tetris	0.668	1.565	4.811	13.393	12.012
Average	4.357	117.505	162.609	995.452	776.630

Notes: The average data for the last row is the average runtime for 6 benchmarks for each protection scheme.

overhead of 252% for VMProtect. We also observed that the code size is mainly determined by the size of the protected code region. The larger the target code to be protected, the large the code size will be. Among all the evaluation benchmarks, the `aescrypt` has a greater code bloat when it is obfuscated by VMProtect-Maximum-protection. Overall, our approach does not significantly the code size of the protected code segments when compared to the commercial counterparts.

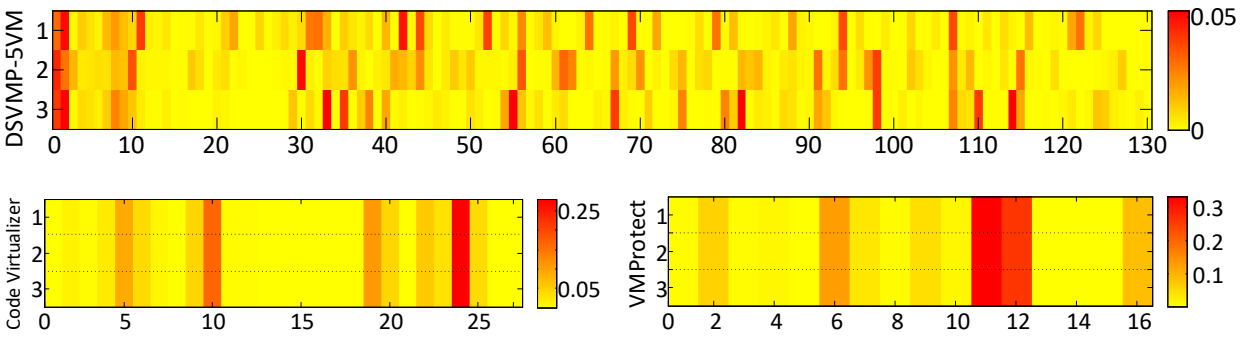


Figure 16: Calling frequencies for handlers across runs per scheme for `md5`. The x-axis shows the number of handlers observed, and the y-axis represents three individual runs.

9.4.2. Runtime overhead

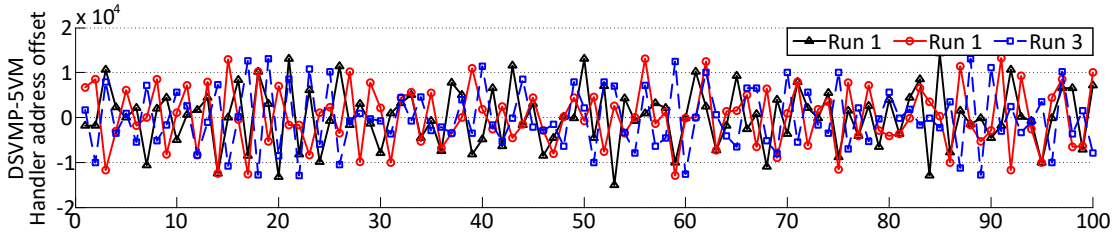
The average runtime overhead of the three schemes is given in Figure 15. This diagram shows that the runtime overhead of DSVMP and VMProtect-Maximum-speed are comparable, which is smaller than CV and VMProtect-Maximum-protection. Code protected under CV and VMProtect-Maximum-protection has the most expensive runtime overhead, which on average is higher than DSVMP and VMProtect-Maximum-speed. Detailed experimental data are shown in Table 5. Specifically, the average runtime overhead brings by CV and VMProtect-Maximum-protection are 7 and 9 times larger than DSVMP-5VM respectively.

9.4.3. Diversity evaluation

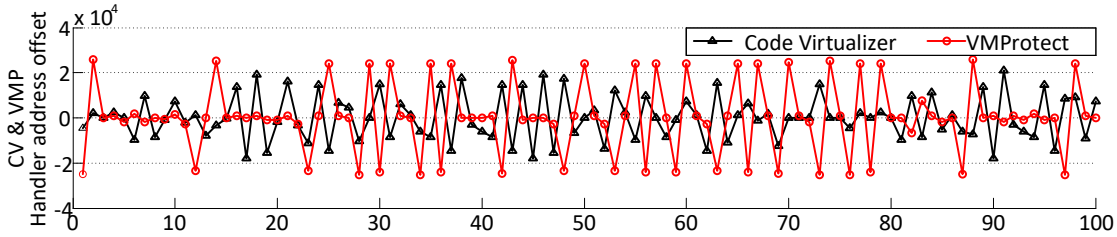
Recall that one of the design goals of DSVMP is to increase the diversity of program execution. To evaluate this design goal, we record how frequent a handler is called across different runs per protection scheme. This experiment was performed on the `md5` application. We ran the obfuscated program three times and used a 5-VM configuration for DSVMP. The results are shown in Figure 16. As can be seen from the heat maps, CV and VMP use fewer handlers than DSVMP; and the calling frequency of each handler remains unchanged across different runs. By contrast, when using DSVMP, the handler frequency changes across runs, resulting in different 'hot' handlers across runs. This example shows that DSVMP can increase the diversity of program execution when compared to CV and VMP, because it uses more handlers and the calling frequency of handlers varies across runs.

Intuitively, the stronger the variation of the handler address offsets across runs, the stronger the non-deterministic behavior the program will exhibit across runs. To collect the dynamic addresses information, we used the Intel Pin [19] binary instrumentation tool instrument the code to gather the handler addresses during runtime. This experiment was performed on the `md5` application and we ran the obfuscated application three times.

Figure 17 (a) shows that the handler address offsets change across runs when using DSVMP. By contrast, CV and VMP (Figure 17 (b)) use a static set of handlers, and the handler address offsets remain unchanged across runs. This experiment shows that code protected by DSVMP exhibits stronger non-deterministic runtime behavior when compared to CV and VMP. This is because the set of handlers used to interpret the virtual instructions can change across difficult runs. This dynamic feature makes it harder for an attacker to use



(a) Code protected by DSVMP



(b) Code protected by CV and VMProtect

Figure 17: The address offset of the scheduled handlers in time order during different runs for `md5` protected under different schemes. The x-axis shows the part of the handler's scheduling at different times. Where applications protected by Code Virtualizer and VMProtect, always follows a single and fixed handlers execution sequence during different runs, so there's only one curve.

the knowledge gathered during the previous runs to perform reverse engineering, because the handler sets to use may change in future runs.

To sum up, compared with two commercial tools protection programs, which always follow a fixed execution path for the same input across different runs, the program protected by DSVMP follows dynamically changing execution paths in different runs.

9.4.4. User study

We follow a similar evaluation methodology described in [20] to conduct a small scale user study to evaluate the strengthen of our protection scheme. Our user study involved 15 students from the host institution. Among the 15 students, there are 2 senior PhD students and 13 postgraduate (Master) students, of which 7 students are female and 8 are males. The students were studying a computer science degree in cyber security at the time this experiment was conducted. Our participants all have hands-on experience on software reverse engineering. This experiment was approved by the research ethics board (REB) of the host institution.

In this experiment, all participants acted as an attacker. They were asked to reverse engineer the `md5` application, which has been obfuscated by three code protection schemes: DSVMP, CV and VMP. In this experiment, we use a 5-VM configuration for DSVMP. The participants tried to accomplish three tasks described as follows. Each participant was given 72 hours to accomplish a task.

- Task 1: Find the entry point address of the VM interpreter;

Table 6: The number of participants who have successfully accomplished a task.

Code Obfuscation Schemes	Task 1	Task 2	Task 3
DSVMP-5VM	11	3	3
CV	11	10	10
VMP	11	8	8

- Task 2: Given the entry point address of the VM interpreter, find the address of the dispatcher;
- Task 3: Given the address of the dispatcher, find out what handlers have been executed during runtime; and record the handler addresses.

It is to note that these tasks represent the essential steps that an attacker must perform in order to reverse engineer a code region protected under a VM-based code obfuscated scheme. Therefore, the fewer people success in a task, the stronger the protection a scheme will provide. We also remark that these tasks are relatively simple, because the protected code region only contains a handful number of native instructions; and accomplish these tasks may not lead to a successful attack, because an attacker still needs to recover the functionalities of the target code. Nonetheless, this experiment allows us to compare the protection strengthen of our approach against commercial counterparts.

Table 6 shows how many participants have successfully accomplished a task. Intuitively, the fewer participants could accomplish a task under a protection scheme, the stronger protection the scheme has provided. Task 1, finding the entry point address of the VM interpreter, is trivial to our participants (who already have hands-on experience on VM-based code obfuscation). Most of our participants could do so for all schemes. Tasks 2 and 3 appear to be harder. When using Dsvmp, there are only three participants managed to accomplish them. This is because the dynamic scheduling structure employed by Dsvmp makes it difficult to trace the addresses of the dispatcher and the handlers. To perform tasks 2 and 3 on CV and VMP seems to be easier. As we can see from the table, 10 participants could successfully finish these tasks when using CV, and 8 participants were able to do so when using VMP. This experiment confirms that Dsvmp indeed increases the cost of reverse engineering when compared to CV and VMP.

To investigate these results further we recorded the correct dispatcher and handler addresses collected by the different participants in task 2 and 3, respectively. The results are shown in Table 7. Each data item of this table is a two-tuple, (N_D, n), where N_D is the number of the correct dispatcher addresses obtained in task 2, and n represents the number of the handler addresses recorded in task 3. Experimental results show that CV and VMP only have a single dispatcher, i.e. the tuple for CV and VMP in Table 7 is (1, 14630429) and (1, 2143262). This means that once the dispatcher address is located, it would be easy to extract the entire handler set. For Dsvmp, however, there can be multiple dispatchers (5 in this example). Because our participants had not been told that there may be multiple dispatchers, most of the them thought they had completed the task when locating the first dispatcher address. As a result, only 3 participants have successfully located all the dispatch-

Table 7: The dispatcher and handler addresses collected by different participants

Schemes	P1	P2	P3	P4	P5	P6
DSVMP-5VM	(3, 1057219)	(5, 2463571)	(1, 24037)	(1, 89692)	(0, 0)	(2, 991895)
CV	(1, 14630429)	(1, 14630429)	(1, 14630429)	(1, 14630429)	(0, 0)	(1, 14630429)
VMP	(1, 2143262)	(1, 2143262)	(1, 2143262)	(0, 0)	(0, 0)	(1, 2143262)

Schemes	P7	P8	P9	P10	P11
DSVMP-5VM	(1, 88805)	(1, 285717)	(2, 477916)	(5, 2463571)	(5, 2463571)
CV	(1, 14630429)	(1, 14630429)	(1, 14630429)	(1, 14630429)	(1, 14630429)
VMP	(1, 2143262)	(0, 0)	(1, 2143262)	(1, 2143262)	(1, 2143262)

This table shows the data of the 11 participants (P1-P11) who have successfully found the VM entry address in task 1. Each data item of this table is a two-tuple, (N_D,n), where N_D is the number of the correct dispatcher addresses obtained in task 2, and n represents the number of the handler addresses recorded in task 3. Number "0" in a tuple means that the participant did not get the data or got the wrong data.

er addresses (see task 2 in Table 6). As a consequence, most of them failed to locate the complete set of handlers. Moreover, since each dispatcher is randomly selected at run time by DSVMP, the handler sequences collected by the same dispatcher addresses are different across runs. This mechanism further increase the difficulty of code reverse engineering.

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 [21], packers [22, 23], are used to protect software against attacks based on disassembly and static analysis. There are also other code protection techniques like code obfuscation [24], control flow and data flow obfuscation [25, 26, 27], 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. In recent years, there are many different code protection programs has been constantly put forward. Some research is devoted to CFI (Control Flow Integrity) protection, for example, Zhang *et al.* [28] and van der Veen *et al.* [29] provide fine-grained CFI systems against modern control flow hijacking attacks based on ROP (Return-Oriented Programming) and more advanced code reuse attacks [30]. And some studies consider Code Randomization protection, such as Stephen *et al.* presents a practical, fine-grained code randomization defense, called Readactor, resilient to both static and dynamic ROP attacks [31].

This paper focus on researching the code virtualization protection, and there is a growing interest in using it to protect software from malicious reverse engineering. Similar to our code virtualization approach is the work conducted by the following studies.

Fang *et al.* [5] proposed an algorithm of multi-stage software obfuscation method. Their approach iteratively transforms the critical code region several times with different interpretation methods to improve security. Adversaries will need to crack all intermediate results to figure out the structure of original code. A similar, Yang *et al.* [6] 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. The multi-stage and nested interpretation process, however, is bound to bring expensive time overhead.

Averbuch *et al.* [32, 33] 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. But it requires hardware support, and the decryption key is stored in the CPU and the attacker cannot get it, so it can effectively hinder the attacker's reverse analysis.

Wang *et al.* [7] proposed a protection scheme to increase the time diversity of protected code regions to resist dynamic analysis. 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. However, once generated, these sub paths are determined and the number is limited, so this approach only provides limited time diversity.

In the meantime, code analysis and deobfuscation techniques are constantly being introduced. Representative techniques such as Symbolic and concolic execution and taint analysis [34, 35, 36]. And Shoshitaishvili *et al.* [37] presents a binary analysis framework *angr*. Their work presents a systematized implementation of analysis techniques that proposed in the past, which allows other researchers to compose them and develop new approaches. Some of the common methods for analysing VM-based protection are as follows, Coogan *et al.* [38] 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. It is often used to analyze Malware, due to the malicious code will inevitably interact with the system. Sharif *et al.* [39] used dynamic data-flow and taint analysis to identify data and extract the syntactic and semantic information about the bytecode instructions. Yadegari *et al.* [40], by tracking the flow of inputs values, and then use semantics-preserving code transformations to simplify the logic of the instructions. These approaches, however, cannot restore the structure of the original code completely, because the analysis process is only for a dynamically executed sequence of instructions and does not cover all branches, and it depends on the results of the taint analysis. Therefore, they are usually performed and analyzed several times with different input to obtain a better structure information.

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 [41].

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 six 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 partially 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); the UK Engineering and Physical Sciences Research Council under grants EP/M01567X/1 (SANDeRs) and EP/M015793/1 (DIVIDEND); and a Royal Society International Collaboration Grant (IE161012).

References

[1] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, T. Xing, G. Ye, J. Zhang, Z. Wang, Exploiting dynamic scheduling for vm-based code obfuscation, in: 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), IEEE, 2016.

[2] Themida, http://www.oreans.com/themida.php.

[3] Code virtualizer, http://www.oreans.com/codevirtualizer.php.

[4] Vmprotect software. vmprotect, http://vmpsoft.com/.

[5] 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.

[6] M. Yang, L. S. Huang, Software protection scheme via nested virtual machine, Journal of Chinese Computer Systems 32 (2) (2011) 237–241.

[7] 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.

[8] 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.

[9] N. Falliere, P. Fitzgerald, E. Chien, Inside the jaws of trojan, Tech. rep., Clampi. Technical report, Symantec Corp (2009).

[10] C. Collberg, The case for dynamic digital asset protection techniques, Department of Computer Science, University of Arizona (2011) 1–5.

[11] R. Rolles, Unpacking virtualization obfuscators, in: 3rd USENIX Workshop on Offensive Technologies.(WOOT), 2009.
[12] C. S. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation-tools for software protection, IEEE Transactions on Software Engineering 28 (8) (2002) 735–746.
[13] Ida pro, https://www.hex-rays.com/index.shtml.
[14] Ollydbg, http://www.ollydbg.de/.
[15] Sysinternals suite, https://technet.microsoft.com/en-us/sysinternals/bb842062.
[16] Peering inside the PE: A tour of the Win32 portable executable file format, https://msdn.microsoft.com/en-us/magazine/ms809762.aspx.
[17] B. Blietz, A. Tyagi, Software tamper resistance through dynamic program monitoring, in: Digital Rights Management. Technologies, Issues, Challenges and Systems, 2006, pp. 146–163.
[18] Astar and tetris source code., https://github.com/MGKKY/AStar-and-Tetris.
[19] Pin, dynamic binary instrumentation tool, https://software.intel.com/en-us/articles/pintool.
[20] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques, Empirical Software Engineering 19 (4) (2014) 1040–1074.
[21] 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.
[22] Execryptor, http://www.strongbit.com/execryptor.asp.
[23] Upx, http://upx.sourceforge.net/.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] M. Zhang, R. Sekar, Control flow integrity for cots binaries., in: USENIX Security Symposium, 2013, pp. 337–352.
[29] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, C. Giuffrida, A tough call: Mitigating advanced code-reuse attacks at the binary level, in: Security and Privacy (S&P), 2016 IEEE Symposium on, IEEE, 2016, pp. 934–953.
[30] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, T. Holz, Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications, in: Security and Privacy (S&P), 2015 IEEE Symposium on, IEEE, 2015, pp. 745–762.
[31] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, M. Franz, Readactor: Practical code randomization resilient to memory disclosure, in: Security and Privacy (S&P), 2015 IEEE Symposium on, IEEE, 2015, pp. 763–780.
[32] 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.
[33] A. Averbuch, M. Kiperberg, N. J. Zaidenberg, Truly-protect: An efficient vm-based software protection, IEEE Systems Journal 7 (3) (2013) 455–466.
[34] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, A. D. Keromytis, A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware., in: NDSS, 2012.
[35] M. Balliu, M. Dam, R. Guanciale, Automating information flow analysis of low level code, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014,

pp. 1080–1091.
[36] B. Yadegari, S. Debray, Symbolic execution of obfuscated code, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 732–744.
[37] S. Yan, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, Sok: (state of) the art of war: Offensive techniques in binary analysis, in: IEEE Symposium on Security and Privacy (S&P), 2016, pp. 138–157.
[38] 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.
[39] M. Sharif, A. Lanzi, J. Giffin, W. Lee, Automatic reverse engineering of malware emulators, in: 30th IEEE Symposium on Security and Privacy (S&P), IEEE, 2009, pp. 94–109.
[40] B. Yadegari, B. Johannesmeyer, B. Whitely, S. Debray, A generic approach to automatic deobfuscation of executable code, in: IEEE Symposium on Security and Privacy (S&P), 2015, pp. 674–691.
[41] 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.