Exploiting Dynamic Instruction Scheduling for VM-Based Code Obfuscation

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

***Abstract*—Code virtualization building upon virtual machine (VM) technologies is emerging as a viable method for implementing code obfuscation to protect programs against unauthorized analysis. State-of-the-art VM-based protection approaches use a fixed scheduling structure where the program follows a single, static execution path for the same input. Such approaches, however, are vulnerable to dynamic cumulative attacks 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 protecting software against dynamic cumulative attacks. 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 commercial applications. Experimental results show that DSVMP performs well in terms of security and runtime overhead.**

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

I. INTRODUCTION

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

Code virtualization building upon a virtual machine (VM) is emerging as a promising way for implementing code obfuscation[1], [2], [3], [4], [5], [6], [7]. The idea of VM-based protection is to replace the native machine code with virtual bytecodes which will then be translated into native machine code at runtime using a set of bytecode handlers. 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 parts: a dispatcher that determines which virtual instruction (i.e. bytecode) is ready for execution, and an instruction handler that translates the bytecode into native machine code to be executed on the underlying hardware. This process replaces the original program instructions by using virtual bytecode, allowing developers to conceal the purpose or logic of sensitive part of the program.

Prior work on VM-based software protection has primarily focused on making the bytecode more complicate, but they only define a fixed set of bytecode instructions and use one scheduler. Such approaches rely on the assumption that the scheduler and the bytecode instruction set are difficult to be cracked in most practical runtime environments. However, research has shown that is an unreliable assumption [8]. In fact, an adversary can easily crack a software product by reusing the knowledge extracted from other applications protected with the same set of VM tools. This is known as the *dynamic cumulative attacking method*. To solve the problem, it is vital important to introduce a certain degree of diversity into program execution [9].

This paper presents DSVMP, a novel VM-based code protection scheme to addresses the problem of dynamic cumulative attacks. Our key insight is that it will be much more difficult for the attacker to analyze the code if the program behaves differently in different runs. We achieve this by introducing rich diversity into the program execution and bytecode interpretation. To increase diversity of the program behavior, DSVMP exploits a flexible, multi-dispatched scheme for code scheduling 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 behaves differently. Our carefully designed scheme ensures that the program will produces a deterministic 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 how the program algorithm works. As a result, this significantly increases the cost for reading, writing and reverse-engineering a program.

Uncertainty and diversity are keys for deterring dynamic

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

cumulative attacks. DSVMP brings together two techniques to increase uncertainty and diversity of program behaviour. Firstly, to provide a certain degree of uncertainty, DSVMP provides multiple handlers (that are implemented using d-ifferent algorithms and data structures) for each bytecode instruction. Handlers for a particular bytecode instruction 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, the program execution path is likely to be different in different runs. Secondly, DSVMP employs a multi-VM scheme so that different 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 reusable knowledge extracted from other software products (as different products are likely to be protected by different bytecodes instructions and VM implementations).

This paper makes the following contributions:

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

II. BACKGROUND

VM-based code obfuscation transform native instructions of the protected code regions to virtual instructions. The virtual instructions are encoded into bytecodes which will be translated native machine code at runtime by bytecode handlers. Indeed, a new VM section is inserted to the end of target program and remaining code is kept intact, thus entry point of the protected code is redirected into VM section. In general, classic VM bytecode execution is stack based style, and need for a virtual context (VMcontext), which usually is a block of memory that contains the virtual registers and flags. The return value of each bytecode is saved in virtual registers, for next bytecode execution. When entering the VM, the native context is stored in the VMcontext to initialize the virtual context, and when exiting the VM, native context will be restored based on the virtual context. The VM interpreter is the core of the VM, which consists of a dispatcher and the handling procedures of bytecode instructions (handlers). The main loop that iterates through three phases for every instruction: decode, dispatch and execute[10]. Werewolf or assassin, dispatcher fetch a bytecode, then decode and dispatch a handler to interpret it. Obviously, the adversary can crack the original software easily if he understands the virtual interpreter and decodes mapping between two instruction sets (binary codes and bytecodes).

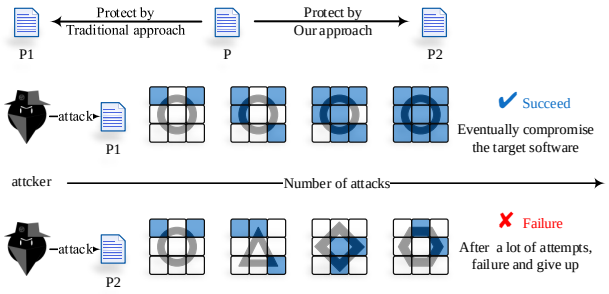

Fig. 1. Diversity affects effectiveness of attack. Software crack like puzzle game, the dark small square represents the reusable attack knowledge. A single structure can be defeated by cumulative attack easily, but diversity of structure can against such attacks effectively.

Dynamic cumulative attack Figure1 illustrates how an attacker can perform attack by reusing knowledge extracted from the previous runs of the same application or other applications (that are protected using the same VM scheme).In general, there is a positive correlation that the more times the attack is, the more reusable attack knowledge will be mastered, and eventually compromises the target software. If a software shows dynamically variant execution paths in different run times, which can destroy this kind of positive correlation. Our work is to achieve this purpose.

III. THE ATTACK MODEL

We assume that the adversary has an executable program of the target software, and he can run it in the malicious host environment[11]. The adversary has necessary attack technology, he can access memory or registers and trace instructions, and even can change the original instructions and control flows by using some static and dynamic analysis tools, like ”IDA”[12], ”Ollydbg”[13], ”Sysinternals suite”[14], etc. His goal is to completely reverse the internal implementation of target program, and then use that knowledge to grab benefits for his own.

The classical approach to reverse engineer a VM-protected program is consists of following steps[8], [15]. First, reverse engineer the virtual interpreter which consists of dispatcher and handling procedures, the attacker needs to locate them and analyzes what they do. Then, use this information to collect the individual bytecode instructions and its mapping relationship with the handling procedures. Finally, recover original internal logic implementation of the target program. Once an attacker has analyzed some information about the structure, he could reuse it for continually analyzing. The cumulative attack is the attacker accumulates these reusable attack knowledges through repeated reverse analysis, and eventually compromise the target software. In our attack model, the adversary is a skilled attacker, he can use any analysis tools and be familiar with the implementation mechanisms of VM, and he follows the steps described above while reverse engineering a VM-obfuscated program.

IV. DSVMP Overview

DSVMP is a fine-grained code virtualized software protection system that can provide a resolution to make control flow in different ways every time.

Following the common practice in VM-based protection, DSVMP also focus on protecting the critical code that has significant value, which can improve the performance. DSVMP's infrastructure also includes the dispatcher to fetch the bytecode instructions. The implementation process of DSVMP at a high level goes through the following steps, as shown in Figure2.

Step1: Locate the target code segment, and use the disassembling engine to get the x86 instructions.

Step2: Map x86 instructions to virtual instructions based on VIS (Virtual Instruction Set).

Step3: Different from the traditional VM, DSVMP improved the handler and dispatcher that add the structural control unit for them, which are used to realize the uncertainty structure and multi-VM scheduling.

Step4: Generate multiple sets of handlers and dispatchers. We use the *VMNum* (>=1) and *DisNum* (>=1) to indicate the number of VM and dispatcher respectively. The original handlers set will be obfuscated *VMNum* times by using the deformation engine, and disrupt the sequences of each set of handlers. Then we get *VMNum* sets of equivalent but different forms of handlers. Similarly, generate *DisNum* dispatchers which have the same function but, with various forms.

Step5: Generate *2*VMNum* sets of driver-data. Map VI and handlers then generates *VMNum* sets driver-data (which is composed of the handler's serial number and its parameters.). And the offset value (that between the current handler and next handler) creates the rest of *VMNum* sets driver-data.

Step6: Construct the multiple VM, which is composed of the *VMNum* sets of handlers and *2*VMNum* sets of driver-data.

Step7: Insert a new section at the end of protected file, which contains *VMNum* VMs and other VM components such as dispatcher, VMcontext, VMinit and VMexit etc.

V. System Design For DSVMP

This paper introduces DSVMP, which distribute a unique structure at every runtimes. We outlined a practical approach that provides tailor-made driver-data. The diversity of DSVMP depends on two parts of the work: the diversified scheduling structure and the Multi-VM. In this section, we will present the details of our solution.

A. Diversified Scheduling Structure

As a core of the VM, the scheduling structure should be considered first. There are three kinds of VM-based scheduling structure, the chain structure of scheduling, the centralized scheduling structure and the multi-dispatcher scheduling structure. Our system is based on the multi-dispatcher scheduling structure. Further, we redesign the atom handler and use two sets of driver-data to improve this structure.

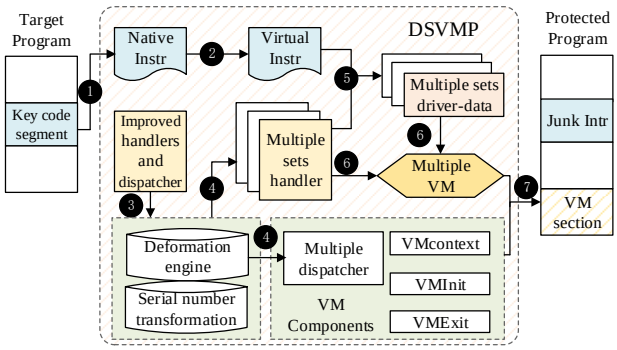

Fig. 2. Overview of DSVMP. First get the VI from critical codes, and using parameters to generate multiple sets of equivalent but form different handlers. Then generate the corresponding driver-data and constitute the multi-VM. Finally add Multi-VM and other VM components to a new section and fill the junk instructions into the original key code section.

1) Redesign the atom handler: The initial atom handler's structure is one set of functional components that not be obfuscated, which are scheduled for execution by the dispatcher. Traditionally, the dispatcher fetches the driver-data to decrypt and compute out the sequences of the handler. According to the sequences, one set of handlers is executed definitely. If the adversary traces the program repeatedly, it is not difficult to figure out the specific bytecode instructions.

To overcome this challenge, we redesign the initial handler to create a new handler. There is an example to illustrate how we to improve and enhance the handler structure. We will insert a control unit at the end of the handler, and it used to do a random selection, return to the dispatcher or directly execute the next handler. The new handler is shown below. LOADS instruction is used to fetch the extra added parameter that is the offset value of two adjacent handlers.

```
lods byte/word/dword ptr ds:[esi]
... ...
push eax
rdtsc           //------------------------
mov ecx,2
div ecx         //structure control unit
cmp edx,0
jz label        //------------------------
lods dword ptr ds:[esi]
... ...         //directly to next handler
add dword ptr ds:[edi+48],eax
jmp dword ptr ds:[edi+48]
label: push ebx//------------------------
div bl
movzx eax,AH    //or return to dispatcher
add eax,9dH
```

2) Create two sets of driver-data: Only the control unit is not enough. We also need one new set of driver-data to cooperate. As the Figure2 shows, the driver-data is of great importance, which determines the handler's execution sequences. However, one set driver-data is difficult to provide random sequences and the one-to-one relationship between

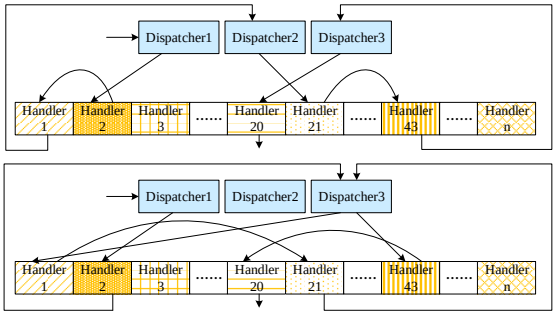

Fig. 3. Two types of possible scheduling structure. In the figure, the handlers' execution sequence is: H2→H1→H21→H43→H20. We can see that there was a big difference between the results of different runtime.

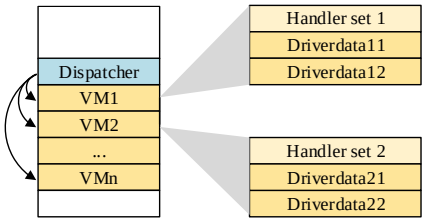

Fig. 4. The structure of multiple VM. Each VM contains one set of unique handlers and two sets of driver-data, *DriverDataSetN1* and *DriverDataSetN2*.

handler and driver-data will be clearly identified easily. To solve the above-mentioned problem, we propose two sets driver-data solution.

DriverData1 and *DriverData2* will be introduced for describing the creating process. The *DriverData1* and usual driver-data are the same and it is for dispatcher, which is composed of the handler's serial number and its parameters. The first data in *DriverData2* is the handler serial number, the rest of it are composed the offset value between two adjacent handlers and parameters. If the control unit chooses to execute the next handler, it will fetch the corresponding offset value from the *DriverData2*. They are all stored encrypted.

3) Show the random structure: The method mentioned above can simply implement the diversity of the program execution path. Once the previous handler finished, it has the option between returning dispatcher to fetch *DriverData1* and accessing the *DriverData2* to fetch the offset value. Obviously, if there are more dispatchers, it will have a lot more choice.

In order to explain how this scheduling structure works at runtime, we illustrative our results by some examples. If the key code segment needs five handlers and three dispatchers, there will be 4^4 sets structures to be chosen. Figure3 shows two possible uncertainty structures. The execution sequences of basic block are very different, and the first attack knowledge on the control flow will be useless to the next attack directly.

B. Multi-VM switch

In the default settings, VM-based obfuscation protects the key code regions with a single VM (SVM) structure. We put forward the Multi-VM (MVM) for user to choose and which contains multiple sets of equivalent but different forms of handlers and driver-data.

Typically, SVM only generates one set of driver-data and handlers. No matter how complex its structure is the relationship of handler and bytecodes is fixed. This will be operable weakness for the adversary. In contrast, our MVM has a dynamic mechanism and it will provide the different handler serial numbers for different instances of protection, for the same instance, the same function handler will also have a different serial number and structure. In fact, every execution will random switch driver-data from multiple VM, and it would show different mappings between the handler and driver-data. As the adversary, if he cannot find this mapping relationship, he wouldn't restore the integrally original code and data. Next we illustrate the details of multi-VM.

1) Generate multiple Handler sets: HAS as an original handler set that consists of *n* handlers, it will be obfuscated for *VMNum* times for building multi-VM, then get multiple handler sets and which are semantic equivalence but have different forms. However, all of the equivalent handlers have the same serial number. This is a kind of directly mapping relationship and is detrimental to the security of the system. So further disrupt the serial number of the handler in order to enhance the effect of obfuscation, and the relationship of these equivalent handlers in different sets should be: $HAS_1(i) \Leftrightarrow HAS_2(j) \Leftrightarrow \ldots \Leftrightarrow HAS_{vm}(k)$, $1 \leqslant i, j, k \leqslant n$, it will produce variety mapping relationship of handler and VI. There are various methods to upset the handler's serial number, we take a simple method that all the serial number plus a uniform random number, then regard the result MOD *m* as the new handler's serial number.

Given the above design, we need to consider one practical issue that is the balance of the security and performance. Depending on the design of the previous section, we need to generate two driver-data sets for each set of handler due to the variety of mapping relationship. So the more the embedded VM number is, the more secure the protected code and the bigger the file size will be.

2) Schedulability of Multi-VM: The former dispatcher structure function only fetches the driver-data to decrypt it and dispatch handler. To schedule the multiple VM, we need to improve the dispatcher structure. The dispatcher will make choice in multiple VM at each runtime. Firstly, we calculate the driver-data offset value between the running VM and the switching VM. Then adjust the value of ESI (register), the pointer of the new driver-data address according to the offset value. All of the dispatchers and handlers will run as the same form after being improved. They all possess the random choice function. It is hard to distinguish two types structure that can enhance the concealment of dispatchers.

Figure 4 shows the multiple sets VM drive structure, every execution wills random switch driver-data from these VM, if the result is switch to VM2 from VM1, the dispatcher need to adjust value of ESI point to *DriverDataSet21* of VM2 and continue to fetch driver-data to dispatch handler. On the contrary, continue to fetch the driver-data from VM1's *DriverDataSet11* and continue to execute.

Ins1	Ins2	Ins3	Ins4
move 0x08 load move 0x04 store	move 0x04 load load 0x03 sub store move 0x04 store	load 0x40103b	ret

TABLE I
RESULTS OF THE NATIVE INSTRUCTIONS VIRTUALIZATION.

VI. CASE STUDY WITH DSVMP

After discussing the designs of DSVMP, we will give an example to illustrate how the system operates. There is a little piece of code. "00401036" and "00401038" is the instruction address. "STARTSDK" and "ENDSDK" are the Start flag and End flag of the key code segment.

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

A. Process of protection

The first, positioning and disassemble the key code segment, then we will get two x86 instructions. "mov eax, ebx" and "sub eax, 03", add two instructions, "push 40103B" and "ret", in order to jump back to execute the rest of code after exits the DSVMP. Then convert x86 instructions to VI follow the transformation rules of design in advance, and the results as shown in Table I. Our system is stack-based, so, "load" instructions are for pushing operands into stack, and "store" instructions are for popping results out of the stack and store it into the virtual context.

We set the embedded VM as 2 (VM1 and VM2). Transforming the improved handlers using the deformation engine, generating two sets of equivalent but form different handlers, then randomly disturb the serial number and get two new sets of handlers, HAS1 and HAS2. Then, VI was encoded into bytecodes which we call the driver-data. According to HAS1, generates the *DriverDataSet11*, which composed of the handler serial number and parameters, illustrated in Figure 5. Then built on *DriverDataSet11*, calculate the offset value and get the D*riverDataSet12*. Similarly, According to VM2's HAS2 also can get two sets of driver-data *DriverDataSet21* and *DriverDataSet22*.

Then encrypt four sets of driver-data, the encryption key is kept by register EBX and it will be modified after each encryption. Here the *DriverDataSet11* and *DriverDataSet12* use the same key *Key1*, and *DriverDataSet21* and *DriverDataSet22* use another key *Key2*, to ensure that the encryption and decryption are symmetrical when switching the VM. Finally, fill key code segments with junk instructions, and create a new section attached to the end of the original file, which contains two sets of handlers, four sets of driver-data, ten dispatchers and other VM components, VMcontext, VMInit and VMExit, etc.

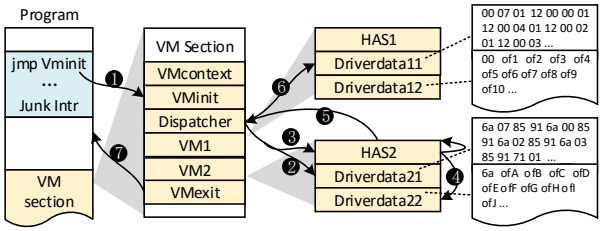

Fig. 5. The execution process of protected program. Embedded VM number is 2, and each VM contains two sets of driver-data and one set of handlers.

B. Process of execution

Through the above process, software will have diversity of execution path. Specific process is shown in Figure 5.

Step1: At runtime, upon executing the key code segment, a "jmp VMInit" instruction will transfer the control to the VMInit. VMInit saves the host context and initializes the virtual context, containing the initialization of the VMID, we assume it is VM2.

Step2: Next, dispatcher starts to work and it fetches a bytecode from the *DriverDataSet21* decodes it gets "6a". Jump to and execute "0x6aHandler", the next bytecode "07" is its parameters.

Step3: Start executing the structure control unit when "0x6aHandler" executed over. It randomly selects to continue execution "0x85Handler" or return to the dispatcher. If choose return to the dispatcher, move on to step 5.

Step4: The "0x6aHandler" fetches a bytecode from *DriverDataSet21*, decodes it and get offset value. According to this value jumps to the "0x85Handler" continues to execution. Move on to step 3.

Step5: There are 10 dispatcher embedded. Randomly select one and jump to execute.

Step6: Execute the selected dispatcher. Select one VM randomly and complete switches VM by adjusting the pointer value of the register ESI. Fetches a bytecode from driver-data, decodes it and get the handler serial number. Then jump to the handler and back to Step3.

Step7: Step3 and Step4 are iterated until complete the all bytecodes. Then execute VMexit, which restore and jump back to native context continue to execute the left code.

VII. SECURITY ANALYSIS

The protected software will show the variant execution paths in different run time, related analyses are as follows.

A. Runlevel software structure similarity analysis

In order to gather statistics on the number of all possible execution structure when the instance runs, we assume that the number of dispatcher is 10. Here, the number of the dispatcher is an optional protective option. In the case of Section VI, the length of the *DriverDataSet11* and *DriverDataSet21* are all 103 byte, contains a total of 78 handler serial numbers. Remove the last handler because of it will directly exit the VM when it runs over. All the rest of the 77 handlers have 11

kinds of choice. Each handler can directly jump to the next handler, or return to one of the 10 dispatcher.

So there is 11^{77} possible execution paths. So the probability that appears the same control flow structure in a continuous operation is

$$P = \frac{1}{11^{77}}$$

Here just has one VM, if we embed 5 VM, each dispatcher also can have 5 kinds of choice. The more number of dispatcher and VM is, the more number of control flow structure will be. So we can know that the DSVMP can make software with very low run level structure similarity.

B. The code level software structure similarity analysis

If all of the software protected by the same protection system have the exactly same features, the attacker can use these features to crack any software protected by this tool. In order to show that our system can resilient such attacks, we calculate the similarity of two software structures to assess the difficulty of attacker crack software (protected by DSVMP).

Blietz et al.[16] present some concept to describe the structural features of control flow, like branch number, cycle index, node nesting level and so on. On the basis of existing literature, we present some indices for structural diversity, details are as follows.

`NodeNum`, is the number of basic blocks.

`BranchNum`, is the number of basic blocks which the last instruction is the conditional jump instruction.

$DR(Vi)$, is the basic block degree relationship of *Vi*, and $DR(Vi) = D_{in}(Vi) + D_{out}(Vi)$, $D_{out}\,(Vi)$ refers to the out-degree and $D_{in}\,(Vi)$ refers to the in-degree.

Similarly, $DF(Vi)$ is the data flow relationship of basic block *Vi*, used to indicate the frequency of *Vi*'s information exchange. $DF\,(Vi) = Flow_{in}(Vi) + Flow_{out}(Vi)$, and $Flow_{in}$ is the number of reading instruction in *Vi*, $Flow_{out}$ is the number of writing instruction in *Vi*.

As shown in Table II, the second column is the key code segment. Their structures are all very simple that only have one basic block and there is no branch block, the structure similarity is close to 100%. We get A' and B' by using DSVMP, the related information of the structure is also as shown in Table II. We use a formula to calculate the software structure information of A' and B' respectively, the results are as follows.

$$\begin{aligned} SInfor_{A'} &= NodeNum_{A'} + BranchNum_{A'} \\ &\quad + \sum_{i=0}^{i<n} (DR(i) + DF(i)) \\ &= 23 + 5 + (DR(0) + DR(1) + ... + DR(22) \\ &\quad + DF(0) + DF(1) + ... + DF(22)) \\ &= 28 + (46 + 18) \;=\; 92 \end{aligned}$$

$$\begin{aligned} SInfor_{B'} &= NodeNum_{B'} + BranchNum_{B'} \\ &\quad + \sum_{i=0}^{i<m} (DR(i) + DF(i)) \\ &= 48 + 9 + (DR(0) + DR(1) + ... + DR(47) \\ &\quad + DF(0) + DF(1) + ... + DF(47)) \\ &= 57 + (96 + 36) \;=\; 189 \end{aligned}$$

Basic info of target program				Info of protected-software		
prog.	**key code**	**Dis Num**	**VM Num**	**prog.**	**Node Num**	**Branch Num**
A	mov eax,ebx sub eax,03	5	5	A'	23	5
B	pop eax add eax,ebx	10	10	B'	48	9

TABLE II
THE RELEVANT INFORMATION ABOUT THE PROGRAM.

From the $SInfor_{A'}$ and $SInfor_{B'}$, we can calculate the software structure similarity $SDiff$ of A' and B'.

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

Thus it can be seen the software structure similarity between A' and B' is only 34.5%, DSVMP has greatly increased the difficulty that the attacker perform the dynamic attack by using control flow and data flow information.

VIII. PERFORMANCE EVALUATION

In this section, we present a detailed discussion about experiment and cost analysis, and then evaluate the effectiveness of DSVMP based on the experimental data in detail.

A. Security experimental evaluation

We verify the security of DSVMP by using an attack experiment, and the number of VM and dispatcher in the instance is set at 3 and 5.

Using the tool "IDA"[12] to debug the instances,and then locate the start tag of key code segment and get the control flow graph of the program after into the virtual machine, but the last one basic block ended in an indirect jump instruction, "IDA" can not automatic acquisition the target address, so it is incomplete. We need to collect the dynamic instructions, and then manually connect the basic block to draw a complete control flow graph. By utilizing taint analysis and control dependencies to simplify and extract the software structure information. But the control flow structure has changed when we try to analyze the software again, previous attack experience is useless, we can only take time again to draw complete control flow graph in current state.

Then we use a tool "OllyDbg"[13] to perform the dynamic debugging, and then locate the dispatcher. Register ESI points to the driver-data, defining the ESI for tainted data and track its transmission process, we can find that the dispatcher fetchs the bytecode and stores it in register EAX after decryption, these are the serial number of the handler. Collect all values that scheduled by each dispatcher, as follows:

```
00 0A 12 15 37 64 00 0A 12 89 0B 02 57 36
78 9A 8E 65 0C 13 25 32 11 24 0A 12 24 8F
4E 35 34 01 02 07 12 24 09 2A 31 44 05 01
0F 12 34 52 35 09 ... ...
```

But these serial numbers are composed of multiple VM, and it is incomplete because some handlers did not schedule

through the dispatcher. So it is difficult to distinguish the relationship between these handler serial numbers and VMID. DSVMP embed multiple sets of handler in virtual interpreter, the attacker wants to analyze these handler completely is very difficult. Even it can be fully analyzed, we still unable to crack the software, because we don't know the relationship of handler serial number and set of handler.

B. Runtime cost evaluation

We evaluate the performance impact of DSVMP on a PC with Intel® Core^{TM} 2 Duo processor at 3.00GHz with 4.00GB of RAM. The operating system environment is Windows 7. We selected four test cases. They respectively are md5.exe[17], aescrypt.exe[18], bcrypt.exe[19], and gzip.exe[20], they are all used to process a file (test.jpg) of the size 763KB. Table III shows the basic information of these target programs. The 3th column indicates the critical function and the 4th column indicates the number of instructions in these critical functions. The numbers of instructions executed in the critical functions while processing the test.jpg file is shown in the last column.

1) Impact on File Size and Runtime Performance: For each target program, we use the prototype of DSVMP to protect it for 5 times, each time using a different number of VM. Figure 6 (a) shows the impact on file size of DSVMP with multi-VM protection. Since each VM has two sets of driver-data and one set of handlers, the file size of the protected program increases as the number of VM increases. Besides, exist a positive correlation between the sizes of and the number of critical instructions, so "aescrypt" has the fastest increasing of file size, "bcrypt" is slowest.

To evaluate the runtime overhead that DSVMP introduces, we use each programs to process the "test.jpg" file for 10 times. Then we calculate the average runtime overhead per critical instruction and the results are shown in Figure 6 (b). The runtime overhead increases as the number of VM increases. But 3VM always has a lower overhead than 2VM, this is not just a coincidence, one possible reason is the implementation strategy we use in multi-VM protection. Besides, "aescrypt" has a much higher runtime overhead that other program. Because its critical code more complex and needs more virtual instructions to interpret.

2) Comparison with two commercial VM-based protection tools: To illustrate the applicability of DSVMP, we compare it with two commercial VM protection systems, Code Virtualizer[2] and VMProtect[3]. As we only compare the impact of code virtualization protection, we disable other protection techniques that integrated into VMProtect and CV, e.g., code compression, resource protection, etc. Besides, CV has 24 custom VMs that are devised ahead of protection, and we choose one of moderate complexity and speed, namely the FISH32 (White), to protect. As for DSVMP, it with 2 VMs. Figure 7 (a) shows the impact on file size of three different VM protection systems. CV has the fastest increasing of file size, but the increasing of the size is relatively stable and independent on original program size; this is true for DSVMP and VMProtect as well. Figure 7 (b) introduces the comparison

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

TABLE III
THE BASIC INFORMATION OF THE TARGET PROGRAM.

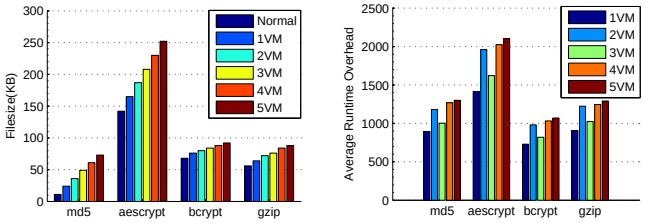

Fig. 6. (a) The impact on file size (KB) of DSVMP that embedded different VM. (b) The average runtime overhead per critical instruction use the different number of VM protection.

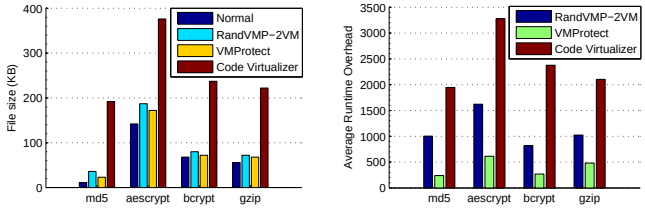

Fig. 7. (a) The comparison of impact on file size (KB) with VMProtect and Code Virtualizer. (b) The comparison of average runtime overhead per dynamically executed critical instruction with VMProtect and Code Virtualizer.

of three VM protection systems on average runtime overhead. CV has the largest runtime overhead while VMProtect the smallest and DSVMP the moderate.

DSVMP improved the atom handler, operations of the control unit will cost some time overhead, and embedded the multiple VM also will increase the volume. In general, our approach gets a better performance.

IX. Related Work

Early works on the binary code protection relied on some simple encryption and obfuscation methods, however, these approaches can only provide limited protection when faced with complex diversified attacks. Typically, junk instructions[21], packers[22], [23], above technology usually are used to resist disassembly and some static analysis. There are code obfuscation[24], control flow and data flow obfuscation[25], [26], [27], etc. which are mainly used to obfuscate the semantic and logical information of the target program. So in practical applications, these approaches are seldom caught alone, and they usually combine with each other to protect an instance.

There is a growing interest in using code virtualization to protect the software from malicious reverse engineering. We've already introduced the general process of classical VM-based protection in section II and some possible attacks in section III. Here are some of the research work focuses on

VM-based protection. Fang et al.[4] proposed a protection scheme that is a multi-stage obfuscation, which transforms the critical code iteratively for many times by using different interpretations to improve security. Yang et al.[5] presents a nested virtual machine, and the adversary has to reverse engineer fully each layer of the interpreter step by step before to the next layer. Averbuch et al.[28] introduces the encryption and decryption technology on the basis of VM-based protection, which uses the AES algorithm and the custom encryption key to encrypt the virtual instructions, at runtime, the first decrypt the virtual instruction and then dispatch a handler to interpret it. Wang et al.[6] put forward the time diversity, which constructs several equivalent but different forms sub paths, and these sub paths will be randomly selected to achieve the diversity at runtime.

DSVMP presents the dynamic instruction scheduling structure to improve security for software: (i) Improving the atom handler, add a structure control unit for it to realize the diversified scheduling. (ii) Embedded multiple VM, at runtime, the VM interpreter will fetch the execution data from multiple sets of driver-data randomly and to further obfuscate the semantics of bytecode instructions. Software diversity is an effective strategy to prevent mass scale exploitation and cracking[29]. Our system by applying the above approach to provide internal diversity further impedes the reverse attack.

X. Conclusion

In this paper, we introduce the details of improved VM-based protection called DSVMP, it proposes improved diversity scheduling structure and multiple VM to mitigate the vulnerability of the classical VM. We show the implementation process of our system, and we also analyze and evaluate that our DSVMP is practical and effective for resilient the dynamic accumulation attacks.

DSVMP also has vulnerability, multi-VM randomly selected by the dispatcher's selection structure, and in MVM, each VM can independently finish the complete function. Although the handler also has a random select structure that improves the concealment of dispatcher, once the adversary locates the dispatcher, they can tamper with the control structure to affect the results of selection, so that MVM will be failure.

We intend to increase the tamper-proof technology into our system. Further, we can add a security thread library, contains of the anti-debug and anti-dump threads etc. We also can improve the virtual instruction set and driver-data to make each VM only contains part of the whole driver-data, so that the adversary has to analyze the different VMs.

Acknowledgment

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

References

[1] "Themida," http://www.oreans.com/themida.php.
[2] "Code virtualizer," http://www.oreans.com/codevirtualizer.php.
[3] "Vmprotect software. vmprotect," http://vmpsoft.com/.
[4] H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage binary code obfuscation using improved virtual machine," in *Information Security*. Springer, 2011, pp. 168–181.
[5] Y. Ming and H. Liusheng, "Software protection scheme via nested virtual machine," *Journal of Chinese Computer Systems*, vol. 32, no. 2, pp. 237–241, 2011.
[6] H. Wang, D. Fang, G. Li, N. An, X. Chen, and Y. Gu, "Tdvmp: Improved virtual machine-based software protection with time diversity," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 2014, p. 4.
[7] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, and Y. Gu, "Nislvmp: Improved virtual machine-based software protection," in *Computational Intelligence and Security (CIS), 2013 9th International Conference on*. IEEE, 2013, pp. 479–483.
[8] N. Falliere, P. Fitzgerald, and E. Chien, "Inside the jaws of trojan," Clampi. Technical report, Symantec Corp, Tech. Rep., 2009.
[9] C. Collberg, "The case for dynamic digital asset protection techniques," *Department of Computer Science, University of Arizona*, pp. 1–5, 2011.
[10] S. Ghosh, J. Hiser, and J. W. Davidson, "Replacement attacks against vm-protected applications," in *ACM SIGPLAN Notices*, vol. 47, no. 7. ACM, 2012, pp. 203–214.
[11] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.
[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] R. Rolles, "Unpacking virtualization obfuscators," in *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.
[16] B. Blietz and A. Tyagi, "Software tamper resistance through dynamic program monitoring," in *Digital Rights Management. Technologies, Issues, Challenges and Systems*. Springer, 2006, pp. 146–163.
[17] "Md5," http://www.fourmilab.ch/md5/.
[18] "Aescrypt," https://www.aescrypt.com/download/.
[19] "bcrypt-blowfish file encryption," http://sourceforge.net/projects/bcrypt/.
[20] "gzip," http://www.gzip.org/.
[21] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 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, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 536–546.
[25] C. Liem, Y. X. Gu, and H. Johnson, "A compiler-based infrastructure for software-protection," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. ACM, 2008, pp. 33–44.
[26] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *Proceedings of the 5th ACM workshop on Digital rights management*. ACM, 2005, pp. 83–92.
[27] V. Balachandran, N. W. Keong, and S. Emmanuel, "Function level control flow obfuscation for software security," in *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*. IEEE, 2014, pp. 133–140.
[28] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "An efficient vm-based software protection," in *Network and System Security (NSS), 2011 5th International Conference on*. IEEE, 2011, pp. 121–128.

[29] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Security and Privacy (S&P), 2014 IEEE Symposium on*. IEEE, 2014, pp. 276–291.