Exploit Dynamic Data Flows to Protect Software Against Semantic Attacks

Kaiyuan Kuang[†], Zhanyong Tang[†]*, Xiaoqing Gong[†], Dingyi Fang[†], Xiaojiang Chen[†], Heng Zhang[†], Jie Liu[‡], Zheng Wang[§]
[†]*School of Information Science and Technology, Northwest University, P.R. China.*
[‡]*Tencent Technology (Shenzhen) LI.* [§]*School of Computing and Communications, Lancaster University, UK.*

***Abstract*—Unauthorized code modification based on reverse engineering is a serious threat for software industry. Virtual machine based code obfuscation is emerging as a powerful technique for software protection. However, the current Virtual machine code protection are vulnerable under semantic attacks which use dynamic profiling to transform an obfuscated program to construct a simpler program that is functionally equivalent to the obfuscated program but easier to analyze. This paper presents DSA-VMP, a novel VM-based code obfuscation technique, to address the issue of semantic attacks. Our design goal is to exploit dynamic data flows to increase the diversity of the program behaviour. Our approach uses multiple bytecode handlers to interpret a single bytecode and hides the logics that determine the program execution path (it is difficult for the attacker to anticipate the program execution flow). These two techniques greatly increase the diversity of the program execution where the protected code regions exhibit a complex data flow across multiple runs, making it harder and more time consuming to trace the program execution through profiling. Our approach is evaluated using a set of real-world applications. Experimental results show that DSA-VMP can well protect software against semantic attacks at the cost of little extra runtime overhead when compared to two commercial VM-based code obfuscation tools.**

***Keywords*-VM-based software protection; Data flow obfuscation; Semantic attack; Data flow analysis**

I. Introduction

Unauthorized code analysis and modification based on reverse engineering is a major concern for software companies. By making the program harder to be traced and analyzed, code obfuscation based on a virtual machine(VM), is a viable means to protect applications from unauthorized code modification [1, 2]. The underlying principle of VM-based code obfuscation is to replace the native instructions with virtual bytecodes which will then be translated into native instructions at runtime by a VM interpreter. This forces the attacker to move from a familiar environment of native instruction set (e.g. x86) into an unfamiliar computing environment. As a result, such techniques can significantly increase the cost of attacks. There are a number of VM-based code obfuscation approaches have been proposed[1–6].

Despite much progress has made for VM-based code obfuscation, it remains an open problem to protect software against semantic code transformation, a technique that translates an obfuscated program to produce a simpler program that is functionally equivalent to the obfuscated code but easier to analyze. To extract the program semantics, existing semantic code transformation techniques [7–9] all rely on data flow analysis to understand how the virtual instructions are scheduled. Existing VM-based code obfuscation cannot effectively protect software against semantic attacks, because traditional virtual machine protection methods pay their attention to improve the security of virtual machine structure, but ignore the security of data flow information and that will cause the execution of the program and data flow information to be easily obtained by an attacker. The key to address this problem is to introduce a certain degree of complexity and diversity into the program data flows.

This paper presents DSA-VMP (Defending Semantic Attacks-Virtual Machine Protection), a novel VM-based code obfuscation system, to protect software against semantic attacks. The design methodology of DSA-VMP is to increase the data flow complexity of the protected code region. It will be more difficult for the attacker to analyze the code if the data flow has dynamic and diverse behavior. We employ multiple separated processes (two in our current implementation) to execute the VM so that much virtual bytecode interpretation is done across multiple processes scheduled. To further increase the diversity and complexity, we also employ multiple procedures (handlers) to translate one bytecode instruction. For the same bytecode, multiple handlers produce semantically equivalent results, but are implemented (or obfuscated) in different ways and follow different program execution paths. During runtime, our VM instruction scheduler randomly selects a handler to translate a virtual instruction to the native code. Since the handlers is chosen randomly at runtime for each bytecode and the implementation of different handlers are different, the dynamic program execution path will likely vary in different runs.

We evaluated DSA-VMP on five widely used real-world applications. Experimental results show that DSA-VMP can successfully protect software against semantic attacks with little extra runtime overhead when compared to two commercial VM-based code obfuscation tools: VMProtect[1] and Code Virtualizer[2]. The contributions presented in this paper are summarized as follows:

- Employ multiple processes to provide stronger protection for VM-based code obfuscation;
- Exploit dynamic data flows to protect software against

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

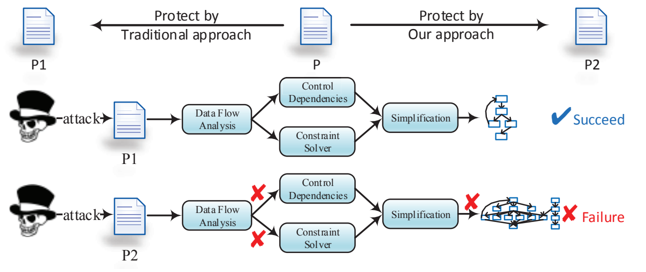


Figure 1: The effect of defending semantic attack. After the software protected by the method of anti-semantic attack, the attacker cannot obtain the Control Dependence and Constraint Solver of the program through the data flow analysis, finally the attacker get a wrong or inaccurate control flow graph.

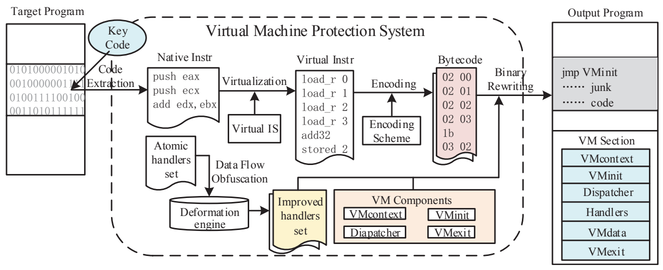


Figure 2: The overview of DSA-VMP system framework. The basic idea of the virtual machine software protection method is to transform the protected native instructions(Intel x86) to virtual instructions which will then be encoded into bytecodes(VMdata) and interpreted by VM interpreter, into which the data flow obfuscation are also introduced to resist semantic attack.

semantic attacks.

II. Background

VM based software protection method is to transform the native code to virtual instructions which will be encoded into bytecodes (VMdata). The VM interpreter will interpret these bytecodes, it follows the decode-dispatch approach and consists of a Dispatcher and the handling procedures of bytecode (handlers).

As illustrated in the Figure 1, the key of semantic attack is to analyze the control flow and data flow of the program. Firstly, the attackers use test input generation tools to explore the space of execution paths, and then use the taint analysis and symbolic execution to analyze the accessibility of the path, and finally obtain the Control Dependence and Constraint Solver of the program. Since the number of the execution paths maybe large when the program is complicated, the attacker need to simplify the original program control flow graph and the internal logical structure. Using these well-known techniques, the core algorithm of the program can be reversed. As stated above, this kind of attack method is not limited to the structure of the virtual machine, so the program protected based VM will cause potential cracking risks. This paper mainly aims at this kind of attack method to put forward a kind of the virtual machine protection method that can defense the semantic attack.

III. Attack Model

In our attack model, we assume that the attacker has an executable program of the target VM-protected software, and he can run it in the malicious host environment [10]. The attacker has full access to the system, he can execute the program at any time and take advantages of any static and dynamic analysis tools [11–13] to help trace and analyze instructions, monitor registers and process memory, and even change instruction bytes and control flows at runtime, etc. We assume that the attacker completely understand the virtual machine protection principle and the structure of the virtual machine. The ultimate goal of the attacker is to completely reverse the program's internal structure and logic.

IV. Design Details

The target program of DSA-VMP is the executable file on Windows platforms (.exe, .dll , etc.), and it mainly focuses on the virtualization protection of critical code in the target program. Figure 2 shows the overview of the DSA-VMP.

A. *DSA-VMP Fundamental Principles*

The procedure for the DSA-VMP to protect the target program as follows:

The key code are firstly extracted and converted into native code (x86 instructions) using a disassembler. Then convert the native x86 instruction into a virtual instruction on the basis of ensuring semantic equivalence. Next encode the virtual instruction to generate the corresponding bytecode instruction. All of the virtual instruction will be stored in the protected program in the form of bytecodes. Then obfuscate the handlers set, the data stream of the Handlers set is obfuscated by using the deformation engine. Finally reconstruct the target file, embed the handler set, VMdata and other key components of the VM into a new section, and fill the critical code with junk instruction, and then redirect the entry of the protected code region to the VM section.

B. *The execution procedure of the protected software*

The execution procedure of the protected software is shown in Figure 3. Specific steps are as follows:

An added jump instruction jumps to VMinit when the program runs on original cirtical code. Then initialize the VMcontext, and map the actual register context to VMcontext (**Step 1**). Execute Dispatcher and Extract the handler sequence is obtained after decryption, and an abnormal interruption which is captured by an abnormal capture mechanism in parent process is generated (**Step 2**). When an abnormal interruption is captured by the mechanism, the handling of this interruption will be performed, then check the interruption information address table to determine the address of the interruption (**Step 3 & Step 4**). Then return to the handling mechanism and perform the next step, jump to the corresponding handler to execute according to the obtained address (**Step 5 & Step 6**). After the Handler is executed, return to run Dispatcher repeatedly

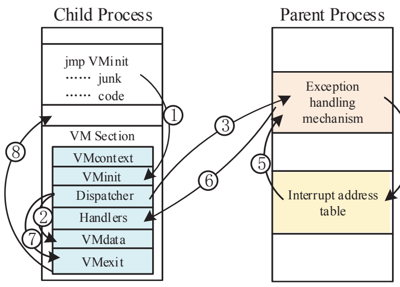
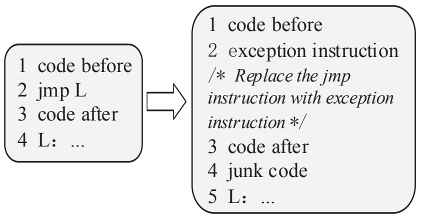


Figure 3: The execution procedure of the protected software.



```
1 code before
2 jmp L
3 code after
4 L: ...
```

```
1 code before
2 exception instruction
/* Replace the jmp
instruction with exception
instruction */
3 code after
4 junk code
5 L: ...
```

Figure 4: An example of a simple hidden predicate.

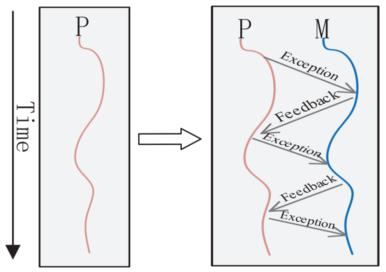


Figure 5: The program execution flow after DSA-VMP protected.

until all the bytecode is explained, and then jump to VMexit (**Step 7**). VMexit restores the virtual register context to the actual register context, and then jump to the end address of the original critical code section, continually running the instruction following the critical code section (**Step 8**).

C. DSA-VMP key implementation

1) Data flow obfuscation of atomic handler: The bytecode is ultimately interpreted and performed by the handler of the virtual machine, so the obfuscation of handler data flow is crucial. As shown in follows is a normal handler:

```
1  _asm
2    {
3      lods byte ptr ds:[esi]
4      xor al,bl
5      add al,52
6      sub al,0AA
7      xor bl,al
8      movzx eax,al
9      push dword ptr ds:[edi+eax*4]
10   }
```

The mission of this handler is to fetch a bytecode from VMdata (line 3), and decrypt the cryptographic bytecode (line 4-7), then calculate the address of the virtual register according to the bytecode, and push the data of this address into the stack (line 8-9). Here, *esi* points to the starting address of the VMdata, and *edi* points to the starting address of the VMcontext.

When analyzing a program, the attacker first given an input value and marks it as a taint data (eg: to taint marks the bytecodes), when the handler executes the instruction *lods byte ptr ds:[esi]*. The taint will be spread to the register of *al*, and in the subsequent calculation process, the taint will continue to spread. So that the attacker can use the data flow to accurately analyze the layout of the register in the virtual context, and then follow-up analysis of the work.

The data flow obfuscation for the handler can effectively prevent the attacker from using the taint propagation to analysis program. The follows shows a handler which has been obfuscated with the data flow:

```
1  _asm
2    {
3      lods byte ptr ds:[esi]
4      xor al, bl
5      add al, 52
6      sub al,0AA
7      xor bl,al
8      push ebx    //Introduce an untainted register ebx
9      mov bl,0FF
10     inc bl
11     cmp al,bl
12     jnz 10
13     mov al,bl
14     pop ebx
15     movzx eax,al
16     push dword ptr ds:[edi+eax*4]
17   }
```

we have added a register operation about *ebx*, and the role of it is to block the taint propagation path. The obfuscated handler implements the same operation as the normal handler, they are functionally equivalent. In the process of *eax* processing, the self-increment operation of the register *ebx* is inserted, when the value of *ebx* is equal to that of *al*, assign the *ebx* values to *al* (line 10-13), finally bleach the taint data, blocking the taint propagation path. Basis on which we can perform various transforming operation to handler to increase the complexity of the program data flow, and it is difficult for the attacker to collect the instruction information in the program execution path.

2) Resistance symbol execution analysis: When the attacker analyze program by the symbolic execution, it first locate to the predicate information in the program, then symbolizes the instruction to infer the accessibility of the path, and finally constructs the control flow information of the program. So the key of resistance symbolic execution is to hide the predicate information in the program.

In the design of the virtual machine, after execution the handler will have a jump instruction to jump back to Dispatcher, loop the decode and dispatch until finished all the bytecodes. Based on this, we will hide jump instruction after each handler, at the same time, randomly add some predicate information to construct fake branches, make the control flow graph constructed by the attacker is either incomplete or is filled up with false branch structure information.

We hide the predicate information in the program using the exception mechanism as shown in Figure 4, the specified method is to modify the last jump instruction of the handler into an abnormal instruction to produce an abrupt. When the exception instruction is captured, according to the interrupt address through the lookup table to find out the target address of the jump instruction, and then jump to original target address. Exception instructions are implemented using the commonly used x86 instructions, such as zero division exception, memory access exception and interruption exception. So that the attacker cannot easily find the exception instruction from a large number of normal x86 instructions. Further can construct the exception library, and random select a exception instructions when replacing the original jump instruction to achieve a certain variety.

3) Dual process confusing program: After introducing the exception, there needs an exception handling mechanism to catch exceptions, thus, making it possible to reach the destination address of the original jump instruction without changing functionality of the original code.

There are many forms of exception handling mechanism in the Windows environment, such as structured exception handling (SEH), vectored exception handling (VEH), C++ exception handling mechanism, etc. We handle the exception mechanism with the idea of double process, as shown in Figure 5, P is the original process of the program, M is an exception capture and handling process, process of the mutual cooperation of process P and M complete the function of the original program. Meanwhile adding a certain process for monitoring, real-time monitors process P completes the mission according to the semantics of the source program, so as to avoid the hijacked by an attacker.

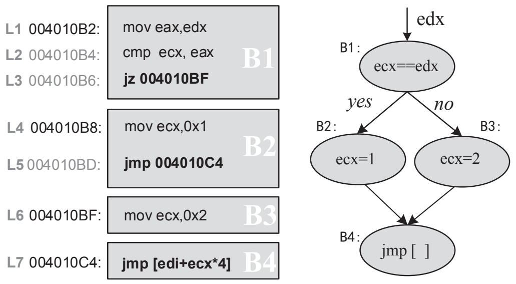
V. Security Evaluation

Collberg and others[14] proposed three indicators to evaluate obfuscating algorithm: strength, flexibility, and the cost. In this section, we will analyze the effectiveness of the protection method, and the next part is mainly on the system overhead assessment

A. Protection Effect Analysis

DSA-VMP is proposed to mainly deal with the present semantic based analysis, attackers usually use the following two techniques, symbolic execution and taint analysis technology, or combination of both. The first step is collect a complete program execution path using the taint analysis, and then do reachability analysis of this path combining symbolic execution, so as to analyze the other path information of the program. But an attacker is unable to collect the complete implementation path of the program through the taint analysis after we protect. Because when a tainted data is marked, as the program executing, the program will bleach the tainted data, thus an attacker cannot collect any path of complete execution. In addition, the protected program hides a large number of predicate information, making it impossible for an attacker to perform an analysis using symbolic execution, that eventually lead to attackers to give up to analysis the program. The following examples are analyzed to illustrate the effectiveness of protection.

Figure 6 shows a code segment, which is divided into the basic block according to the jump instruction, the right is the control dependent relationship corresponding to basic block. It is easy to known that execution result of L3 will determine whether B1 and B2 are executed, it also will determine the


```
L1 004010B2:  mov eax,edx
L2 004010B4:  cmp  ecx, eax              B1
L3 004010B6:  jz 004010BF

L4 004010B8:  mov ecx,0x1                B2
L5 004010BD:  jmp 004010C4

L6 004010BF:  mov ecx,0x2                B3

L7 004010C4:  jmp [edi+ecx*4]            B4
```

Figure 6: The code basic blocks and control flow graph of the program.

Table I: Symbolic execution of binary code.

Num	Assembly code	Symbolic execution process
1	`mov eax,edx`	eax=f_{mov}(var$_1$)= var$_1$
2	`cmp ecx,eax`	f_{cmp}(ecx,var$_1$)
3	`jz 004010BF`	if(ecx-var$_1$ ==0) goto: 004010BF
4	`mov ecx,0x01`	ecx=f_{mov}(0x01)=0x01
5	`jmp 004010C4`	goto:004010C4
6	`mov ecx,0x02`	ecx=f_{mov}(0x02)=0x02
7	`jmp[edi+ecx*4]`	goto:[edi+ecx*4]

register of *ecx*s value, and ultimately will determine which is the destination address the basic block B4 jump to.

When attackers analyze on this program segment, the operation is as follows, use the taint analysis technology to mark *edx* as a tainted data, instruction L1 propagate taint to *ecx*, *ecx* is also marked as the taint data. At this point, if the *ecx* is equal to *eax*, then execute the jump instruction between L3 and L6, and then execute L7 jump to the corresponding destination address. This process collect an execution path: B1$\rightarrow$B2$\rightarrow$B4. And then use the execution path combine with symbolic execution to deduce the rest of execution path. As shown in the table I, instruction 1 transfers variable var$_1$ to *eax* via the *mov* instruction, instruction 2 compares register *ecx* with variable, instruction 3 judges the compared results and determines to execute instruction 4 or instruction 6, thus the corresponding value of *ecx* is 1 or 2. If the instruction does not jump, then the constraint expression for the execution path is *ecx*-var$_1$!=0, if the expression is satisfied, the execution path is B1$\rightarrow$B3$\rightarrow$B4. Or else change the constraint expression deducing another execution path is B1$\rightarrow$B2$\rightarrow$B4. Eventually get the logical structure of the program.

As shown in Figure 7, it is a program information after protection (here we only analyze added exception instruction), in which we use the instruction exception using simple int3 exception specification. What it is shown in right figure is the block relationship between basic program blocks and does not represent the specific execution process.

Using the same method to carry on the analysis, we found that taint propagates to L2 instruction, when encountered exception interruption of L3 could not continue to propagate downward. In addition, we also added the taint bleaching technology in the program, such as it is shown in section IV-C, further blocking the propagation of pollution source, so the attacker is not able to collect a complete exe-

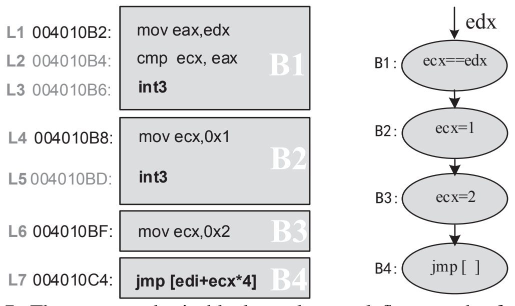


Figure 7: The program basic blocks and control flow graph after hiding predicate information.

Table II: Symbolic execution of the protected program.

Num	Assembly code	Symbolic execution process
1	mov eax,edx	eax=f_{mov}(var$_1$)= var$_1$
2	cmp ecx,eax	f_{cmp}(ecx,var$_1$)
3	int3	Null
4	mov ecx,0x01	ecx=f_{mov}(0x01)=0x01
5	int3	Null
6	mov ecx,0x02	ecx=f_{mov}(0x02)=0x02
7	jmp[edi+ecx*4]	goto:[edi+ecx*4]

cution path information. Table II is the result of performing symbol execution analysis on this basis. We can find that the attackers cannot calculate path constraint expression from symbolic expressions, therefore, it is impossible to further speculate other paths to construct the control logic structure of the internal program.

VI. Experimental Evaluation

We evaluated DSA-VMP on a PC with an 3.3 GHz Intel(R) Core(TM) i3-2120 processor and 4GB of RAM. The PC runs the Windows 7 Pack Service 1 operating system.

A. *Experimental use cases*

In order to test performance overhead of virtualization software protection method of anti-semantic attack, we selected five programs using known algorithm achievement as test cases. They contain a calculator, compression, message transmission, matrix multiplication, recursive algorithm, and to some extend all of them are representative. Details are shown in table III, in the table, I_P represents the number of x86 instructions for the key code segment of the protection program, I_E represents the number of the instructions that actual program execute, and the data is obtained and traced dynamically by Pin[15]. Among them, the I_P of calculator and IpMsg are higher than the I_E of them, and this is because that the branch instructions exist in programs and the programs have not implemented these instructions actually. The I_P of Compress, MatrixMul, Hanoi, are lower than I_E of them, and this is because there are large amount of circulation, recursion instruction, and there will be more instructions are executed during the actual execution.

B. *DSA-VMP system performance analysis*

Using DSA-VMP to protect test programs, we recorded the execution time (average execution time) and file size

Table III: Test case description.

Program	Critical Code	I_P	I_E
Calculator	Windows calculator, Multiplication operation	48	31
Compress	File compression algorithm, Processing 8KB size text file	110	312686
IpMsg	Simple communication tools, Send message algorithm	363	257
MatrixMul	Matrix multiplication algorithm, Computing the 6 order matrix	60	9150
Hanoi	Hanoi algorithm, Enter the plate number is 6	82	2628

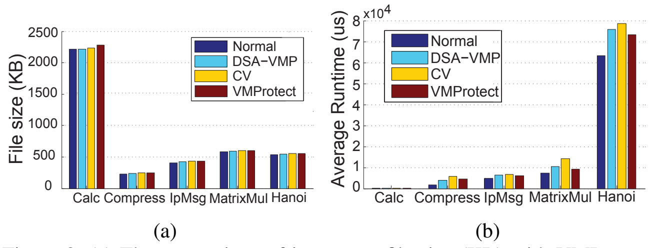


Figure 8: (a) The comparison of impact on file size (KB) with VMProtect and Code Virtualizer. (b) The comparison of average runtime overhead (us) with VMProtect and Code Virtualizer.

Table IV: Average runtime overhead per dynamically executed critical instruction (*µs/per_instr*).

Calculator	Compress	IpMsg	MatrixMul	Hanoi
1.032	0.008	6.124	0.346	4.700

of the original programs and the protected programs. The results are shown in Figure 8.

The impact caused by DSA-VMP on the size of files is mainly because that DSA-VMP adds new virtual machine sections in the protected programs, and in each part of virtual machine, only the bytecode size is not fixed, and the others are fixed, this is no relationship to the test programs. In Windows, the each section of PE files are aligned according to a certain alignment number (0.5kb or 4 kb). So, from the data in Figure 8, we can see that the size of IpMag files increases by 16kb,and other four test programs increases by 8kb.It is because that the number of instructions protected by IpMsg is more than that of other test programs. In the meanwhile, the bytecode produced are relatively large.

We calculated the average consumption time of each execution instruction to represent the performance consumption from system, and the expressions are as follows:

$$Cost_{per\_instr} = (T_A - T_B)/I_E$$

- Cost$_{per\_instr}$: Performance overhead for each instruction(*µs/per_instr*);
- T$_A$: The execution time of the program after the protection;
- T$_B$: The execution time of the original program;
- I$_E$: The number of the instructions that actual program execute.

The table IV shows the average performance consumption caused by DSA-VMP for each x86 instruction in the test programs. From the data in the table, we can see that the performance consumption of the every instruction in

IpMsg are relatively large, which is because that, most of the key code segments in IpMsg are arithmetic instructions and logical operation instructions, and need more handler to explain and execute. As a result, the consumption is larger, and others are mainly based on data transmission instructions. In the meanwhile, it need less executed handler, and the consumption is less as well. In addition, we compared the protection effect of DSA-VMP with two commercial code virtualization protection system, Code Virtualize[2]and VMProtect[1]. The Figure 8 shows the impact on test programs about file size and execution time after the protection of Code Virtualize and VMProtect.

VII. Related Work

Software protection are used to protect the intellectual property encapsulated within software programs from been pirated and modified, by transforming target program into a more obscure and hard-understanding one. In the early years, the protection of the binary code mainly depends on some simple encryption and obfuscation methods, these methods can improve code complexity. Typically, junk instructions[16], equivalent instructions, packers[17, 18], code encryption, above technology usually are used to resist disassembly and some static analysis. There are also other code protection techniques like code obfuscation[19], Control flow and data flow obfuscation[20–22],etc. these protection methods can only provide limited obscurity, so in practical applications, these approaches are seldom caught alone, and they usually combine with each other to protect an instance.

Code virtualization protection in recent years has been used increasingly to protect the code from malicious reverse engineering [1–6]. We've already introduced some of the research work focuses on the protection based on code virtualization in section I, introduced the general process of classical virtual machine software protection method in section II and some possible attacks in section III.

DSA-VMP put forward a method of defending semantic attack to improve VM protection security for software. (i) Improving the atomic handlers, introduce data flow obfuscation to improve the flow of data complexity. (ii) Adopting double process, the virtual machine's structure is distributed in different processes, which makes the implementation of the program more complex and diverse. Our system increases the complexity of the VM's data flow by applying the approach above to resist semantic attack technology.

VIII. Conclusion

This paper presents DSA-VMP, a novel VM-based code protection scheme to deal with the attacks based on semantic analysis. We obfuscates the data flow of the program by using anti-taint techniques in the handler to hide the predicate information. The double process is also introduced to confuse the execution flow of the program. Eventually making the program execution flow and data flow more complex, greatly improve the ability to resist semantic attack. Through theoretical and experimental analysis, the results show that method can resist attacks based on semantic analysis and has little effect on performance.

IX. Acknowledgment

This work was partial supported by projects of the National Natural Science Foundation of China (No. 61672427, No. 61572402), the International Cooperation Foundation of Shaanxi Province, China (No.2015KW-003), the Research Project of Shaanxi Province Department of Education (No. 15JK1734), the Service Special Foundation of Shaanxi Province Department of Education (No.16JF028), the Research Project of NWU, China (No.14NW28). Especially this work was supported by Tencent.

References

[1] "Vmprotect software protection," http://vmpsoft.com/.

[2] "Code virtualizer," http://www.oreans.com/.

[3] H. Fang *et al.*, "Multi-stage binary code obfuscation using improved virtual machine." in *ISC, Xi'an, China*, 2011.

[4] H. Wang, D. Fang *et al.*, "Nislvmp: Improved virtual machine-based software protection," in *CIS*, 2013.

[5] H. Wang *et al.*, "Tdvmp: Improved virtual machine-based software protection with time diversity," in *PPREW*, 2014.

[6] A. Averbuch *et al.*, "Truly-protect: An efficient vm-based software protection," *IEEE Systems Journal*, 2011.

[7] K. Coogan *et al.*, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *CCS*, 2011.

[8] M. Sharif *et al.*, "Automatic reverse engineering of malware emulators," in *S&P*, 2009.

[9] B. Yadegari *et al.*, "A generic approach to automatic deobfuscation of executable code," 2015.

[10] C. S. Collberg *et al.*, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE TSE*, 2002.

[11] "Ida pro," https://www.hex-rays.com/index.shtml.

[12] "Ollydbg," http://www.ollydbg.de/.

[13] "Sysinternals suite," https://technet.microsoft.com/enus/sysinternals/bb842062/.

[14] C. Collberg *et al.*, "A taxonomy of obfuscating transformations," *Department of Computer Science the University of Auckland New Zealand*, 1997.

[15] "Pin tool," https://software.intel.com/en-us/articles/.

[16] C. Linn *et al.*, "Obfuscation of executable code to improve resistance to static disassembly," in *CCS*, 2003.

[17] "Execryptor," http://strongbit.com/execryptor.asp.

[18] "Upx," http://upx.sourceforge.net/.

[19] Z. Wu *et al.*, "Mimimorphism: A new approach to binary code obfuscation," in *CCS*, 2010.

[20] V. Balachandran *et al.*, "Function level control flow obfuscation for software security," in *CISIS*, 2014.

[21] C. Liem *et al.*, "A compiler-based infrastructure for software-protection," in *PLAS*, 2008.

[22] J. Ge *et al.*, "Control flow based obfuscation," in *DRM*, 2005.