Exploit Dynamic Data Flows to Protect Software Against Semantic Attacks

Kaiyuan Kuang†, Zhanyong Tang†*, Xiaoqing Gong†, Dingyi Fang†, Xiaojiang Chen†, Heng Zhang†, Zheng Wang‡
†*School of Information Science and Technology, Northwest University, P.R. China.*
‡*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 code obfuscation techniques 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. Doing so can reduce the effectiveness of dynamic profiling. Our approach using multiple bytecode handlers to interpret a single bytecode and hiding the logics that determine the program execution path (it is difficult for the attacker 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 emerging as a promising way for implementing code obfuscation 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 are easily obtained by an attacker. The key to address this problem is to introduce a certain degree of complexity and diversity to the program data flows, so that it becomes much difficult to track the program execution to reconstruct a semantically equivalent program.

This paper presents DSA-VMP, 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 much more difficult for the attacker to analyze the code if the data flow has dynamic and diverse behavior. We do so by employing multiple (two in our current implementation) separated processes to execute the VM, so that the virtual bytecode interpretation much be done across multiple processes that are scheduled in arbitrary order. To further increase the diversity and complexity, we also employ multiple procedures (handlers) to translate a 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 choice of handlers is randomly determined at runtime for each bytecode and the implementation of different handlers are different, the dynamic program execution path is likely to be different in different runs.

We evaluated DSA-VMP on five widely used real-world applications. Experimental results show that DSA-VMP can successful 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

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

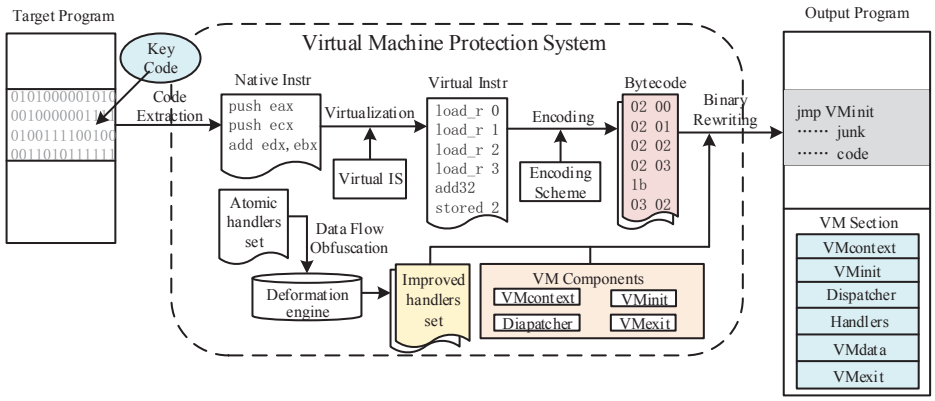


Figure 1: The overview of DSA-VMP system framework. The basic idea of the virtual machine software protection method transform the protected native instructions(Intel x86) to virtual instructions and then virtual instructions will be encoded into bytecodes(VMdata), which will be interpreted by VM interpreter, also introduce data flow obfuscation techniques to the virtual interpreter to resist semantic attack.

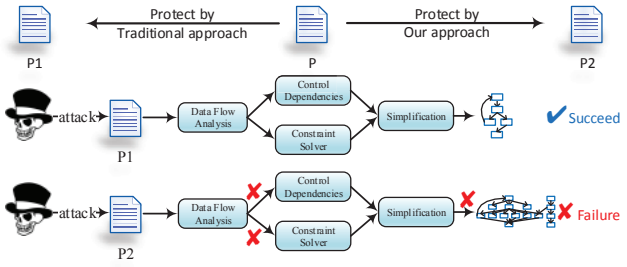


Figure 2: The effect of defending semantic attack. After the software protected by the method of anti-semantic attack, the attacker cannot obtain the Control Dependence and Constraint Solver of the program through the data flow analysis, finally the attacker get a wrong or inaccurate control flow graph.

paper are summarized as follows. It is the first work to

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

II. Background

VM based software protection method is to transform the certain protected code segment to virtual instructions. These virtual instructions could be encoded into bytecodes (VMdata).The VM interpreter will be responsible for interpretation and execution of these byte codes.It follows the decode-dispatch approach, and consists of a Dispatcher and the handling procedures of byte-code instructions (Handlers).

As illustrated in the Figure 2, we can know the semantic attack technologies and the corresponding protection mechanism.The key of semantic attack method is to analyze the control flow and data flow of the program. Firstly, the attacker use test input generation tools to explore the space of execution paths, and then use the dynamic stain analysis and symbolic execution technology to analyze the accessibility of the path, obtain the Control Dependence and Constraint Solver of the program. Since the numbers of the execution path 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 one 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. In our attack model, the attacker uses a variety of techniques to understand the code and reverse the program logic and internal implementation, because of understands the inner logical structure of program is the foundation of tampering, cracking, or re-implementation. we also 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 completely reversing the program's internal structure and logic.

IV. Design Details

The protection program of DSA-VMP is executable file on Windows platforms (.exe, .dll , etc.), it mainly focuses on the virtualization protection of critical code in the target program. Figure 1 shows the overview of the DSA-VMP.

A. DSA-VMP Fundamental Principles

The DSA-VMP works as follows.

Step1: Extracting the key code in target program and disassembling it. SDK(defined by the DSA-VMP) is embedded as start marker and end marker respectively in critical code

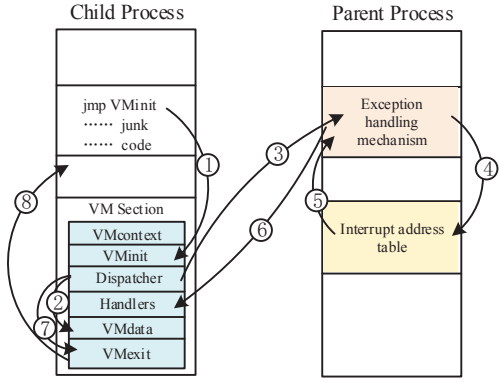


Figure 3: The execution procedure of the protected software.

section. After the PE files are generated, you can locate the start and end address of the critical code section as long as you find the SDK marker, and then to disassemble and get the key code segment.

Step2: Converting the native x86 instructions into a virtual instruction. Make sure the semanteme equivalent, and then convert the x86 instruction into VI according to correspondence between the x86 instruction and VI.

Step3: Encoding the virtual instruction code to generate the corresponding bytecode instructions. Final virtual instruction will be stored in the protected program in the form of bytecode. Encode all virtual instructions, and then obtain the VMdata after the encryption.

Step4: Obfuscation the Handlers set. The data stream of the Handlers set is obfuscated by using the deformation engine.

Step5: Reconstructing the target file. To embed the obfuscated Handler set, VMdata and other key components of the virtual machine into a new section, and fill the critical section with junk instruction, redirect the entry of the protected code region to the VM setion.

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:

Step1: After running the protected software, the 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.

Step2: Execute Dispatcher. Handler sequence is obtained after decryption, and an abnormal interruption which is captured by an abnormal capture mechanism in parent process is generated.

Step3: When an abnormal interruption is captured by the mechanism, the handling of this interruption will be performed.

Step4: Check the information list of the interruption address through the handling mechanism to determine the address of the interruption.

Step5: Return to the handling mechanism, and perform next step.

```
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   }
```

Figure 4: The virtual machine atomic handler without obfuscation.

```
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 a register ebx
                     //that is not stained.
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   }
```

Figure 5: The virtual machine atomic handler after obfuscation.

Step6: Jump to the corresponding Handler and execute according to the obtained address.

Step7: After the Handler is executed, run Dispatcher repeatedly until running all the critical code section, and then jump to VMexit.

Step8: 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.

C. DSA-VMP key technology implementation

1) Data flow obfuscation of atomic Handler: Virtual machine bytecode is ultimately interpreted and performed by the handler of the virtual machine, so the obfuscation of handler data flow is crucial, next, we analyze and explain based on the analysis of an atomic handler.

As shown in Figure 4 that is a handler which has not been obfuscated with the data flow. The mission of atomic handler is to read a bytecode from VMdata (line 3), and then to decrypt the ciphertext bytecode (line 4-7), according to decrypt, the plaintext bytecode compute a virtual address(line 8-9), and push this address into the stack. The starting address of the register of *esi* points to the starting address of VMdata, edi points to the starting address of VMcontext.

The analysis to the atomic handler shows that when analyzing a program, the attacker first gives an input value of the program and marks it as a stain data (eg: to stain the bytecode in VMdata data), when the program execution runs to the handler, it executes the instruction *lods byte ptr ds:[esi]*. The stain will be spread to the register of *al*, and in

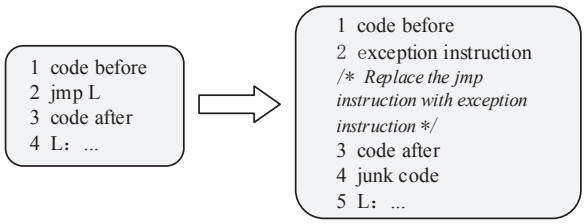


Figure 6: An example of a simple hidden predicate.

the following calculation to the virtual address of context, the stain will last spreading, from which the attacker can use data flow accurately to analyze the layout of the virtual register address in the context, and then to continue the subsequent analysis.

In order to prevent the data flow analysis of virtual machine interpreter handler from trackers, we need to obfuscate the data flow of handler. For prevent the stains from spreading further, we need to cut off the transmission path of stains data and bleach it. Figure 5 shows the data flow of atom handler after obfuscation, from which the adding of a new register *ebx* can be seen, and the role of this register is to prevent the spread of stain path.

As shown in Figure 5 that is a handler obfuscated with data streams, this handler to complete the operation is same as shown in Figure 4, they are functional equivalence.

First, read a bytecode from VMdata(line 3), and then decrypt the bytecode (line 4-7), at this point, a new register *ebx* is introduced to replace the register *al*, whose value has been marked as stain value(line 8-12), the specific operation is the self-add operation of the register *ebx*, when the value of *ebx* is equal to that of *al*, assign the *ebx* values to *al* (line 13), followed by the subsequent instruction of handler to complete the corresponding functions(line 14-17). Through this process, the stain register value is bleached, which prevents the further downward spreading of stain, thereby prevent the attacker from collecting the instruction information in the program execution path. Basis on which we can be perform various transforming operation to handler to increase the complexity of the data flow of the program.

Transform all the handlers through handler's deformation engine to prevent them from analyzing based on data flow.

2) Resistance symbol execution analysis: The key of resistance symbolic execution is to hide the predicate information in the program , because when analyzing the program by using of symbolic execution procedure, the attacker firstly has to locate to the predicate information in the program, and then to symbolize the instruction expressions, in hence to infer the accessibility of the path. Ultimately build a graph of control flow information of the program .

In the design of the virtual machine, after each execution the handler will have a jump instruction to jump back to Dispatcher to fetch a bytecode and decode the execution, loop the execution of fetch code-decode-execute until finished all

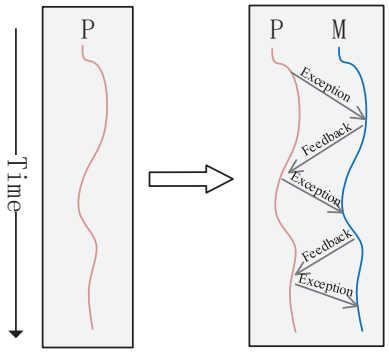


Figure 7: The program execution flow after DSA-VMP protected. The two processes of P and M interaction completes the program function.

the bytecode execution, and then end the loop. Based on that, we will transform the handler, hide every jump instruction of the handler, meanwhile, adding some handler predicate information randomly, construct a fake branch to further confuse the attacker. when analyzing, 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 by using the abnormal mechanism as shown in Figure 6, the specified method is to modify the last jump instruction of the handler in the virtual machine into an abnormal instruction to produce an abrupt. When the abnormal instruction is captured, search the address where the abnormal instruction happens, find out the target address of the jump instruction, and then jump to achieve the original purpose of the jump.

Abnormal instruction design adopt the common x86 instruction to do so with a certain covert action, so an attacker can not easily find the abnormal instruction from a large number of normal x86 instructions in the analyzing the program instructions. First, construct the abnormal instruction library base when designing, and randomly selected abnormal instruction to replace, as shown in Table I is some abnormal instruction of x86, such as zero division abnormal, memory access abnormal and interruption abnormal. And the iteration change of abnormal instruction makes the abnormal instructions various.

Table I: Exception instruction types and examples.

Exception Types	**Examples**
Interrupt	`Int3 /*Int3 exception*/`
Zero Divisor	`mov ebx,0` `div ebx /*Divisor is zero */`
Write Memory	`mov byte ptr ds:[0x00150000],al` `/*Memory address 0x00150000 is read-only section*/`
.....	.....

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

There are many forms of abnormal 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 thinking, as shown in Figure 7, P is the source of the implementation process, M is an exception handling process to capture, 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 completing the mission according to the semantics of the source program, so as to avoid the hijacked by an attacker. compared to the original program when there is a process P only, increasing the complexity of the execution flow of the program makes it possible for an attacker to be found during tracking, in order to make certain protections such as interrupt program, send signal warnings, etc.

V. Security Evaluation

A. Experiment and Analysis

Collberg and others[14] proposed three indicators to evaluation obfuscating algorithm: strength, flexibility, and the cost. Based on Collberg evaluation indicators, this paper combining the characteristics of virtual machines and data flows obfuscation, qualitatively estimating the virtualization software protection with semantic attack-resistant.

Handling data flows obfuscation from three aspects. 1) Data flow obfuscation for atomic handler, introducing anti-taint analysis technique by transferring the handler, make an attacker cannot trace the tainted data propagation paths, generate error, increase the difficulty of analyzing. 2) The predicate information hid introduce the exception mechanism, the instruction constitute exception code is common instructions in a program, such as arithmetic instructions, memory access instructions, and so on. Symbolic execution is difficult to accurately identify exception codes from a large number of common instructions, thus increase the complexity of the symbolic execution locate the path branching point. 3) Introduce double process execution, increases the complexity of program execution flow, makes a single execution flow into the communication between two processes, meanwhile have the function of monitoring main process execution with the control flow as well, has some anti-debugging mechanism as well, having greatly increased the difficulty of the attacker's analysis.

Through comprehensive data flows obfuscation virtual software protection with anti-semantic attack has a high level of protection, and is not easy to be analyzed by attackers.

B. Protection Effect Analysis

DSA-VMP is proposed to mainly deal with the present semantic based analysis techniques, if an attacker want to use semantic based method to reverse software typically attack using two techniques, i.e., symbolic execution and taint analysis technology, or combination of the two techniques. The first step is collecting a complete program execution path used the mothod of taint marks. and then do reachability analysis of this path combining symbolic execution technique, so as to analyze the other path information in 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 execution, in addition, the protected program hiding a large number of predicate information. This makes it impossible for an attacker to perform an analysis using symbolic execution, eventually lead to attackers to give up to analysis the program. The following examples are analyzed to illustrate the effectiveness of protection.

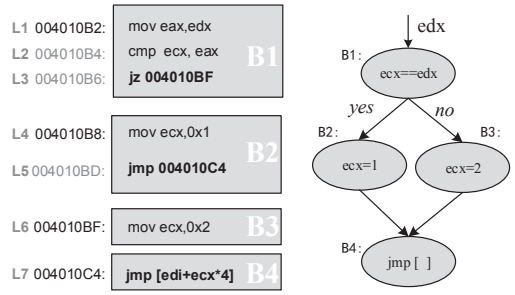


Figure 8: The code basic blocks and control flow graph of the program.

Table II: 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]

Figure 8 shows a code segment, according to the jump instruction it is divided into the basic block, the left is the basic block that has been divided, the right is the control dependent relationship corresponding to basic block. Analysis the code segment can know whether L3 executing or not determine the basic block B2 or B3 executing, also determine the register of *ecx*'s value, and ultimately determine which is the destination address the basic block B4 jump to.

When attackers analysis on this program segment, first use the taint analysis technology to gain the program data flow information, according to the data flow information collect a program's execution path, and then do path reachability analysis use symbolic execution to construct the internal of program's control dependent relationships, 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

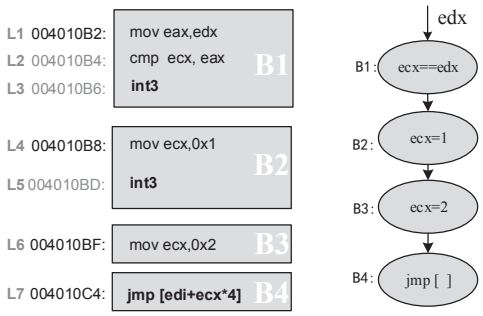


Figure 9: The program basic blocks and control flow graph after hiding predicate information.

marked as the taint data. At this point, if the *ecx* is equal to *eax*, 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→B2→B4. And then use the execution path combine with symbolic execution technology to deduce the rest of execution path.

As shown in the above table II, the instruction 1 via the instruction of *mov* transfer variable var_1 to *eax*, instruction 2 compare register *ecx* and variable, instruction 3 judge the compared results and determine execute the instruction 4 or 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→B3→B4. or else change the constraint expression deducing another execution path is B1→B2→B4. Eventually get the logical structure of the program.

After our system protection, the same method is used for analysis. The analysis processes as follows. As shown in Figure 9, it is the program instruction information after protection (here we only analyzing added exception instruction, and anti-taint analysis is shown in Figure 2), 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 implementation process.

When using the same method to carry on the analysis, we found that taint propagates to L2 instruction, encountering int3 instruction, and 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 4.3, further blocking the propagation of pollution source, so the attacker is not able to collect a complete execution path information.

As shown in Table III, when using symbolic execution to do analysis again, it can be found, from symbolic execution process, that attackers cannot calculate path constraint expression from symbolic expressions, and reason about other execution paths to construct the control logic structure of the internal program.

Through the analysis of the example above, it can be explained that the role of protection is obvious and effective.

Table III: Symbolic execution of the program after the protection.

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]

Table IV: 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

VI. Experimental Evaluation

A. Experimentation hardware environment

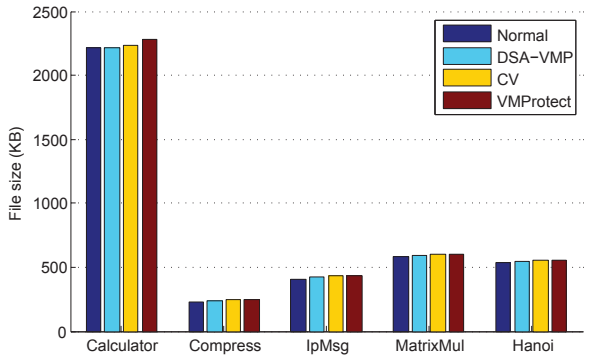
We evaluated DSA-VMP on a PC with an 3.3 GHz Intel(R) Core(TM) i3-2120 processor and 4GB of RAM. The PC runs the Windows 7 Pack Service 1 operating system.

B. Experimental use cases

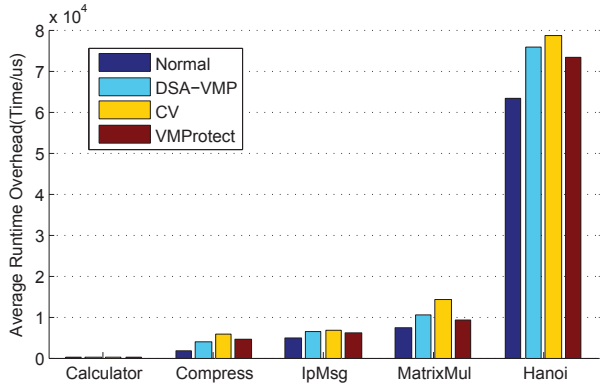
In order to test performance overhead of virtualization software protection method of anti-semantic attack, we selected five kinds of software using known algorithm achievement as test cases. They contain a calculator, compression, message transmission, matrix multiplication, recursive algorithm, and to some extend all of them are representative. Detailed test cases are shown in table IV, 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 did not implement 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.

C. DSA-VMP system performance analysis

Using DSA-VMP to protect test programs, recording the execution time (average execution time) and file size of the original programs and the protected programs. The results are shown in Figure 10.



(a)



(b)

Figure 10: (a) The comparison of impact on file size (KB) with VMProtect and Code Virtualizer. (b) The comparison of average runtime overhead with VMProtect and Code Virtualizer.

The impact caused by DSA-VMP on the size of files is mainly because that DSA-VMP added 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 10, we can see that the size of IpMag files increase by 16kb,and other four test programs increase by 8kb.It is because that the number of instructions protected by IpMsg are more than that of other test programs. In the meanwhile, produced bytecode are relatively large.

We calculated the average consumption value of each execution instruction in program 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.

Table V: 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

The table V 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 virtual machine with two commercial code virtualization protection system, Code Virtualize[2]and VMProtect[1]. The Figure 10 shows the impact on test programs about file size and execution time after the protection of Code Virtualize and VMProtect. By comparing, we can see that the impact caused by DSA-VMP on file is small in comparison with by Code Virtualize and VMProtect, and furthermore the impact caused by Code Virtualize and VMProtect are relatively similar, which is because that the virtual instruction set Handler selected by Code Virtualize and VMProtect are large. In addition, the impact on program execution time caused by DSA-VMP and VMProtect are basically similar, and that caused by Code virtualize is a bit larger, which is because after the protection of the same x86 instructions for different virtual machines, different Handler instructions and the different order of explanation and execution would result in the difference in execution time. But for the users who need high strength protection, the certain consumption of time is acceptable.

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 technology in recent years are more and more be used 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 security for software. (i) Improving the atomic handlers, introduce data flow obfuscation techniques to improve the flow of data complexity. (ii) Adopt 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 by applying the above approach to increase the complexity of the data stream to resist semantic attack technology.

VIII. Conclusion and Future Work

This paper presents DSA-VMP, a novel VM-based code protection scheme to deal with the attacks based on semantic analysis. DSA-VMP mainly from the obfuscation of program data flow by using anti-stain technology for the virtual machine handlers, hiding predicate information in the program, and introduce the concept of double processes obfuscation program execution flow. Eventually making the program execution flow more complex after the protection of the virtual machine, greatly improve the ability to resist semantic attack. Through theoretical and experimental analysis, the results show that the method can resist attacks based on semantic analysis and has little effect on performance.

When implementing anti-taint analysis and anti-symbol execution for handlers in the virtual machine, we can use a variety of anti-taint mechanism and hiding program predicate information method to enhance the strength of resistance based on semantic analysis. Therefore, in the next step of work, we need to design and achieve more anti-stain mechanism and hiding predicate mechanism, and increase the diversity of the protection effect.

References

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

[2] "Code virtualizer: Oreans technology: Software security defined," http://www.oreans.com/codevirtualizer.php.

[3] H. Fang, Y. Wu, S. Wang, and Y. Huang, "Multi-stage binary code obfuscation using improved virtual machine." in *Information Security, International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, 2011, pp. 168–181.

[4] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, and Y. Gu, "Nislvmp: Improved virtual machine-based software protection," in *International Conference on Computational Intelligence & Security*, 2013, pp. 479 – 483.

[5] H. Wang, D. Fang, G. Li, N. An, X. Chen, and Y. Gu, "Tdvmp: Improved virtual machine-based software protection with time diversity," in *ACM Sigplan on Program Protection and Reverse Engineering Workshop*, 2014, pp. 1–9.

[6] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "Truly-protect: An efficient vm-based software protection," *IEEE Systems Journal*, vol. 7, no. 3, pp. 121–128, 2011.

[7] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, Usa, October*, 2011, pp. 275–284.

[8] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *IEEE Symposium on Security and Privacy*, 2009, pp. 94–109.

[9] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," pp. 674–691, 2015.

[10] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 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, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *Department of Computer Science the University of Auckland New Zealand*, 1997.

[15] "Pin-a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[16] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

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

[18] "Upx: The ultimate packer for executables," http://upx.sourceforge.net/.

[19] 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.

[20] V. Balachandran, N. W. Keong, and S. Emmanuel, "Function level control flow obfuscation for software security," in *Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE, 2014, pp. 133–140.

[21] C. Liem, Y. X. Gu, and H. Johnson, "A compiler-based infrastructure for software-protection," in *The Workshop on Programming Languages and Analysis for Security, Plas 2008, Tucson, Az, Usa, June*, 2008, pp. 33–44.

[22] J. Ge, S. Chaudhuri, and A. Tyagi, "Control flow based obfuscation," in *ACM Workshop on Digital Rights Management, Alexandria, Va, Usa, November*, 2005, pp. 83–92.