NISLVMP: An improved VM-based software protection

Huajun Wang\textsuperscript{1,2}, Dingyi Fang\textsuperscript{1,2}, Zhanyong Tang\textsuperscript{1,2}, Xin Song\textsuperscript{1,2}
\textsuperscript{1}School of Information Science and Technology, Northwest University Xi’an 710127, China
\textsuperscript{2}NWU-Irdeto Network-Information Security Joint Laboratory (NISL), Northwest University Xi’an 710127, China
E-mail: hjwang006@163.com, dyf@nwu.edu.cn, http://www.nwu.edu.cn

\textbf{Abstract}—the technology of VM\textit{(virtual-machine)}-based software protection is a hot research field of software protection. Based on researching on the traditional VM-based software protection, we improvement this technology and design new one called NISLVMP, in which there are double register environment, security virtual instruction and more deformation templates. At last, analysis and test show that NISLVMP effectively enhance security of protected software.

\textbf{Keywords}: VM-based software protection; double VM\textit{(virtual)}-Context; Security virtual instruction; deformation engine.

I \textbf{INTRODUCTION}

Software has spread around every corner in the information age, but there are more and more security threats we have to face at the same time. Static and dynamic analysis are been used widely in program comprehension, code analysis, dynamic debugging, and so on. There are many crack tools and technologies. So even if the release of the software generally does not include source code, attackers also can analyze binary machine code to some extent\textsuperscript{[1]}. In this case, software is in White-Box Attack Context\textsuperscript{[2]} surrounded by angers: the software executing process are fully visible to attackers. Although all software can be cracked with enough time, it is need to study software protection to extend its safe period and increase the cost of malicious reverse, and the need for software protection is reaching its height.

Currently, there are some commonly used techniques of software protection, such as serial number, encryption, software watermarking, and shell. Extensive references show that\textsuperscript{[3-6]}, these methods all have certain limitations in practical applications: the serial numbers are only used to protect the program’s entry point; The encrypted program must be decrypted before executed; Software watermarking is applied specially as watermark evidence when software copyright dispute take place and the process of shell is too simple. VM-based software protection\textsuperscript{[7-9]} quickly becomes the hot research field of software protection. T. Maude proposed alternative approach to hide the program interpretation from the user. Barrantes\textsuperscript{[10]} proposed the concept of instruction-set randomization to improve resistance to code-injection attacks. The virtual machine involved in this paper is not system level virtual machine, like VM Ware or Virtual PC. The protection strength has been greatly increased with the development of virtual machine based software protection technique.

This paper is organized to present traditional VM-based software protection in section II. Section III introduces design of double virtual machine context. The virtual security instruction (VSI) is presented in Section IV. Section VI. The security analysis and relative experiments are given in Section VI and Section VII concludes the paper with details of future work.

II \textbf{RELATED WORK}

Once the software has been protected by VM-based software protection technology, software attackers can not decompile the protected codes, then they can't interpret the virtual instructions correctly. Consequently, they have to get the meaning of each code and the interpreting principle of the VM-based software protection.

A. VM-based software protection

In industry, there are some famous virtual machine based software protection, such as code virtulizer\textsuperscript{[11]}, Themida\textsuperscript{[12]} and VMProtect\textsuperscript{[13]}; In academia, there are also some research on this technology. Amir proposed an efficient VM-based software protection\textsuperscript{[14]}. They avoid relying on obscurity and rely only on assumptions about the system itself and on cryptographic measures to develop VM-based conditional access/trusted computing environment, which can defeat Rolles’ attacking method\textsuperscript{[15]}. YANG Ming proposed a software protection scheme via nested virtual machine\textsuperscript{[16]}, thus a reverse engineer has to fully understand the prior layer before he can proceed with the next layer, by this, a high security for software protection is achieved. In the reference \textsuperscript{[17]}, author proposed an idea to obfuscate a software for many times while each time applying different interpretations in order to improve security.

B. Relevant concepts in VM-based software protection

The basic idea of VM-based software protection transforms the key codes into virtual codes, and then destroyed the original instructions, and executing the virtual codes within the virtual-machine interpreter at run-time. Firstly, we will introduce the relevant concepts:

\textbf{Key code segment}: The code with important algorithm or user specified. The virtual machine is mainly to protect these key code segments.

This work was supported by National Natural Science Foundation of China (61070176, 61170218); Cooperation Project of NWU-Irdeto IoT & Info-Sec Joint Lab (2009TR01).
Virtual Instruction (VI): The intermediate results of the instructions which need to be protected after transformation in the VM protection process. These intermediate results are independent of the instruction format, but they don’t exist in the protected program.

VM Interpreter: Virtual machine interpreter is an abstract concept, including VM_Context, Handler and Dispatcher.

VM_Context: It is the environment of the virtual machine interpreting process, and it used to store the values of the registers. When entered into the virtual machine, interpret and execute handlers according to the register value of the VM_Context. At the same time, it will simulate to modify the register values. Finally, before jumping out of the virtual machine, restore the values from the VM_Context to the real registers.

Handler: The essential part of the virtual machine interpreter. It is a group of local instruction fragment used to explain the virtual instruction.

Dispatcher: It mainly used to read Driver_data and jump to execute the corresponding Handler. After execution, the process returns to the dispatcher. Repeat the above steps until using up the Driver_data byte code.

Deformation and Disorder: In general, Handler consists of several assembly instructions, and the prototype is relatively simple. In order to improve the security of the Handler itself, it is need to deform the handlers.

Driver_data: the virtual byte code corresponding to the protected segment. It is generated by the VI. Each Driver_data correspond to a Handler or an argument of the Handler. The virtual machine interpreter reads the byte code and calls the corresponding Handler to implement the function of the protected instructions. This structure exits in the protected program with the virtual machine interpreter.

After explaining the relevant concepts, we will introduce the current virtual machine based software protection technology from the software protection process by virtual machine and the execute process of the protected software.

C. Protection process of VM-based software protection and execution process of protected software

Firstly, it needs to analyze PE file, and also reconstruct PE file after embedded virtual-machine interpreter into software. Secondly, it can transform original instruction into virtual-instruction interpreting by handler. The main protection process is shown in **Fig. 1:**

According to **Fig. 1,** the general steps of the protection process are as below:

1. Analyzing PE file and locating the key code segments;
2. Transforming target (for example, x86) instructions into virtual instructions;
3. Deforming and disordering handlers with the deformation engine;
4. Generating handlers sequence to interpret the virtual instructions;
5. Generating driver data according to the handlers’ sequence;
6. Embedding the driver data, handlers into protected PE file, and then repairing the PE file.

Now, VM had finished protecting software. Then, let’s see how the execution of software protected by VM-software protection, which is shown in **Fig. 2:**

According to **Fig. 2,** the general executing process including 5 steps:

1. When encountering the protected instructions, jump to the main handler of VM;
2. Main handler gets one driver data and decrypts it;
3. According to the decrypted data, get the handler address from jump table and execute the handler, and then jump to Main handler;
4. Repeat step 2 and step 3 until the end of driver data;
5. Jump out of VM and continue executing the codes after the key segment.

D. Existing problems

VM-based software protection had increase the protection strength in some extent. But after researching on these technology deeply, there are also some problems including:

1) **VM_Context is easy to track**

The VM_Context is a temporary structure to record the value of the registers and flags when the virtual instruction is executing. So obtaining the VM_Context is the premise to analyze the software for attackers. And the design of VM_Context directly affects the security of the protected software.
software. Generally, the structure of VM_Context in the traditional VM-based software protection is shown in Fig.3:

```
VM_EAX
VM_EBX
VM_ECX
VM_EDX
VM_EBP
VM_ESI
......
```

Figure 3. Structure of VM_Context in common VM-based software protection

This structure is too simple for attackers to locate the registers and flags in VM_Context by analyzing, and then monitor and track the change of the value of each register and flag to analyze the mean of instructions or behavior of the instruction segments.

2). Single function of virtual instruction

The virtual instruction in traditional VM-based software protection is designed to interpret original instruction set, that is to say, each virtual instruction is valid functional instruction (purely to achieve some certain function). So each virtual instruction interprets the function of the protected procedures section to the attacker. The analysis to each virtual instruction helps the attacker gets a deeper understanding of the protected procedure sections. Some of VM-based software protection are inserted a mass of junk code in the process of protection, but that does not bring too much obstacle to the software attacker.

3). Less deformation templates in deformation engine of handlers

In the protection process of VM-based software protection, deformation engine usually deforms some instructions. General deformation engine usually uses the deformation templates repeatedly. Deformation templates can make one instruction in different forms with the same function. But there are insufficient deformation templates in deform engine, attackers can get the deformation templates of handlers by analyzing several handlers. In addition, combined with the analyzed handler deformation templates, attackers can restore the handler after deformation by manual or automated means. With these experience and results, attackers will design a general attack method against the VM-based software protection.

In order to solve the above problems and improve security strength of protected software, we design new VM-based software protection named NISLVMP. And the improvements will be separately introduced in detail in Section III, IV and V.

III IMPROVEMENT IN VM_CONTEXT

In NISLVMP, double VM_Context structures is designed: VM_Context1 and VM_Context2, and the arrangement of registers in them are different. So values of the registers and flags that virtual instructions using are obtained by associated computing of VM_Context1 and VM_Context2 instead of being obtained from single VM_Context directly. In order to describe better, assume that current virtual instruction uses vm_context in logically. The main designing process is shown in Fig.4:

![Figure 4. Value and assignment of registers in double VM_Context](image)

As shown in Fig.4: In the beginning to protect one instruction, it is need to compute the vm_context value with the value of two VM_Context according to computing rules, and at the end of protecting the instruction, assign to VM_Context1 and VM_Context2 according to vm_context value and corresponding computing rule. So there are three steps in designing double VM_Context process:

**Step1: Design computing rules**

The rules make sure that we can get the right vm_context value after assigning VM_Context1 and VM_Context2, so any algorithm is right which meets the following condition:

<table>
<thead>
<tr>
<th>Algorithm1</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

Then, one effective computing rule includes a pair of algorithm: algorithm1 and algorithm2. And different registers can use different rules. In this paper, we will take some mutual inverse operations for example:

- If \( a \cdot b = c \), then \( a = b \cdot c \);
- If \( a - b = c \), then \( a = b + c \);
- If \( a + b = c \), then \( a = c - b \);

**Step2: The assignment of VM_Context1 and VM_Context2**

In the beginning of protection, the VM_Context1 can be assigned randomly, and assign the VM_Context2 null (and vice versa). When original instructions are interpreted by virtual instructions, the vm_context is known obviously. Then we can compute VM_Context2 value according to associated rules for different registers. Process of assigning of VM_Context2 is shown in Table.1:

**Table.1: The value of VM_Context2 computing rules**

<table>
<thead>
<tr>
<th>Register</th>
<th>vm_context</th>
<th>VM_Context1</th>
<th>VM_Context2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>vm_eax</td>
<td>eax1</td>
<td>eax\cdot eax1</td>
</tr>
<tr>
<td>EBX</td>
<td>vm_ebx</td>
<td>ebx1</td>
<td>ebx\cdot ebx1</td>
</tr>
<tr>
<td>ECX</td>
<td>vm_ecx</td>
<td>ecx1</td>
<td>ecx\cdot ecx1</td>
</tr>
<tr>
<td>EDX</td>
<td>vm_edx</td>
<td>edx1</td>
<td>edx\cdot edx1</td>
</tr>
<tr>
<td>EDI</td>
<td>vm_edi</td>
<td>edi1</td>
<td>edi\cdot edi1</td>
</tr>
<tr>
<td>ESI</td>
<td>vm_esi</td>
<td>esi1</td>
<td>esi\cdot esi1</td>
</tr>
<tr>
<td>EBP</td>
<td>vm_ebp</td>
<td>ebp1</td>
<td>ebp\cdot ebp1</td>
</tr>
</tbody>
</table>
Step 3: The computing of vm_Context
Given VM_Context1 and VM_Context2, the content of vm_context can be ciphered out according to mutual inverse operations given in step 1. Then, we can obtain the algorithms of vm_context, as shown in Table 2:

<table>
<thead>
<tr>
<th>Register</th>
<th>VM_Context1</th>
<th>VM_Context2</th>
<th>vm_context</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>eax1</td>
<td>eax2</td>
<td>eax1^eax2</td>
</tr>
<tr>
<td>EBX</td>
<td>ebx1</td>
<td>ebx2</td>
<td>ebx1+ebx2</td>
</tr>
<tr>
<td>ECX</td>
<td>ecx1</td>
<td>ecx2</td>
<td>ecx2-ecx1</td>
</tr>
<tr>
<td>EDX</td>
<td>edx1</td>
<td>edx2</td>
<td>edx1^edx2</td>
</tr>
<tr>
<td>EDI</td>
<td>edi1</td>
<td>edi2</td>
<td>edi1+edi2</td>
</tr>
<tr>
<td>ESI</td>
<td>esi1</td>
<td>esi2</td>
<td>esi2-esi1</td>
</tr>
<tr>
<td>EBP</td>
<td>ebp1</td>
<td>ebp2</td>
<td>ebp1^ebp2</td>
</tr>
</tbody>
</table>

So in NISLVMP, it is necessary for attackers to extract the value of each vm_context and to determine the corresponding computing rules between them when they analyze the protected key segments, and they also need to compute the final register value according to the corresponding rules.

The double virtual registers determine values of virtual registers, so the execution of handlers depends on the design of double register environment. Handlers interpreting virtual instructions and the register values are depended on the double register environment when designing handlers. Therefore, it can increase the secure strength of VM-based software protection by researching on these three components.

IV IMPROVEMENT IN VIRTUAL INSTRUCTION
In the design of NISLVMP, there are not only the functional virtual instructions, but also the exclusive security virtual instructions (SVI). That is to say, the virtual instructions are not only designed to interpret the original instruction(x86), but also to act a part in secure, and enhance the strength of the VM-based software protection. And now there are two kinds of SVI: Register Rotation Instruction and Anti-Debug Instruction.

RRI (Register Rotation Instruction): To modify the arrangement of the virtual registers in one or two of VM_Context in double VM_Context dynamically in the execution process of the protected software. Then when attacker analyzing the protected software, he should determine the right arrangement of the registers firstly.

ADI (Anti-debug instruction): A group of Anti-debug instructions implements anti-debug function. Some in group monitor whether the protected software is being debugged or tampered, and some withstand for debugging according to the monitoring result, such as killing debugging process, and other ADIs are used to check whether some Anti-Debug Instructions in the group are removed or tampered, if so, it will take attacker into the designed trap. Obviously, the random combination of multiple anti-debugging mechanisms increases the concealment of anti-debugging, and double response can improved the anti-debug ability.

A. RRI (Register Rotation Instruction)

The main principle of RRI is to randomly scramble the register context structure in the VM protection process, and modify the arrangement of the virtual registers dynamically. According to double register context structure of NISLVMP, we design Format of RRI as follow:

$$\text{RRI operand1, operand2}$$

The RRI indicate the rotation type and degree of VM_Context1 and VM_Context2 in NISLVMP register context structure. The operand1 indicates which VM_Context will be rotated, and operand1 ∈ [0, 1] (value 0 indicates VM_Context1 and value 1 indicates VM_Context2), and the operand2 indicates the rotation degree, and operand2 ∈ [0, 7]. What's more, operand1 and operand2 both are determined randomly. Obviously, we can control the VM_Context1 and VM_Context2 rotation at the same time by adding two RRI at the process of the VM protection. In protecting process, a random number is used to decide whether insert RRI instruction or not after each virtual instruction.

The implementation of RRI is easy relatively. VM_Context1 and VM_Context2 are stored in a fixed position in data segment. We design a special handler to interpret the RRI, which can swap and rearrange the register's values according to the value of operand1 and operand2. In the protection process, once insert a RRI, the arrangement of registers in VM_Context after the RRI will change, so it must re-guarantee arrangement of registers in VM_Context before protecting original instructions. That is to say, inserting the RRI cannot affect the original normal function of the protected instructions.

B. ADI (Anti-debug Instructions)

Assume we have designed an anti-debug mechanism, first, slicing the anti-debug functional codes into several small snippets, and then designing ADIs for each snippet. Thus, the set of ADIs implements an integrated anti-debugging function, in which some ADIs are used to monitor, some are used to judge and respond, and others used to mislead attacker to the trap. In NISLVMP, we had designed some sets of ADIs to implement more than one anti-debug mechanism. When protecting software the user can select more than one set of ADIs according to his demand, users can insert ADIs, in which the only requirement is that the ADIs coming from the same set should be inserted in order.

1) Three steps of designing ADI

The designing process of the ADI is shown in Fig.5:
Five Handler Deformation Engine

Handlers interpreting virtual instructions need to be deformed before embedded into protected software. Moreover, once software has been protected, the handlers embedded in the software will not be changed. The design of the deformation engine in NISLVM mainly has two steps:

A. Design the template function of the instruction deformation

The idea to design deformation template function is to accomplish the same function by different assembly instructions. For instance, the value transfer operation between registers can be achieved by "mov reg1, reg2", and can be achieved by the combination of "push reg1" and "pop reg1". When deforming instructions based on deformation template, all it needs to do is calling the corresponding template function.

B. Design the deformation engine based on the templates

Take the original instructions of the handlers as the first input, then deform them by the template function, and select several instructions randomly from the previous result to deform. Iterate this process until getting the number of times the user has set.

The deformation engine working process is shown in Fig. 7:

V. Handler Deformation Engine

Step1. design the anti-debug mechanism. The mechanism mainly includes three parts: action monitoring, judging and responding, and misleading to traps. Action monitoring is to monitor whether the software is being debugged, such as checking parent process, checking software breakpoints and so on. Judging and responding is to make measures according to the monitoring result, usually interrupting the process, bluing the screen or modifying the data of virtual registers etc. If attackers discovered the two parts, the misleading to traps will work, i.e. once the attacker removes the two parts wholly or partially, the third part will mislead the attacker to traps, such as fake execution flow, endless loops and so on.

Step2. slice each part of designed anti-debug mechanism into snippets randomly. Therefore, the snippet can just realize function of one part partially: monitoring, judging or misleading.

Step3. design ADI for each snippet. A set of ADIs can implement a completely anti-debug mechanism. To make these ADIs realizing the function cooperatively, ADIs coming from the same set need an independent registers environment (named AD_Context). The handlers interpreting ADIs do not use the vm_context but the real register Context. So when designing ADIs, it is needed to save the current real register context and restore AD_Context to register context. In the end of the ADIs, it needs to restore real register context and save register context to AD_Context.

2. The cooperation of various anti-debugging mechanisms

When protecting software with NISLVM, users can select more than one kind of anti-debug instruction sets randomly, and independent AD_Context will be added once selecting a mechanism. Now take three sets: FAD(First anti-debug), SAD(Second anti-debug), TAD(Third anti-debug) for example, Fig. 6 is the structure of the protected key segment after inserting more than one anti-debugging mechanisms:

Moreover, these ADIs sets can be used interactively.

V. Handler Deformation Engine

Handlers interpreting virtual instructions need to be deformed before embedded into protected software. Moreover, once software has been protected, the handlers embedded in the software will not be changed. The design of the deformation engine in NISLVM mainly has two steps:

A. Design the template function of the instruction deformation

The idea to design deformation template function is to accomplish the same function by different assembly instructions. For instance, the value transfer operation between registers can be achieved by "mov reg1, reg2", and can be achieved by the combination of "push reg1" and "pop reg1". When deforming instructions based on deformation template, all it needs to do is calling the corresponding template function.

B. Design the deformation engine based on the templates

Take the original instructions of the handlers as the first input, then deform them by the template function, and select several instructions randomly from the previous result to deform. Iterate this process until getting the number of times the user has set.

The deformation engine working process is shown in Fig. 7:

V. Handler Deformation Engine

It is obvious that the core of the deformation engine is the design of the template. The more templates are designed, the better deformation results of the handlers. However, in the process of the handler deformation iteration, we must guarantee the equivalent before and after deformation (including the same results after executing, the same influence on flag register). Deformation templates are designed in NISLVM by two principles:

- Decompose the stack operation instructions;
- Split some arithmetic and logic instructions based on discrete mathematics.

There are no templates that are designed to split the arithmetic and logic instructions based on discrete mathematics in the traditional VM-based software protection. Consequently, combined with the two principles, it will increase the difficulty when attackers analyzing the handlers.

The two type templates will be introduced in the following:

1. Design the deformation template based on decomposing the stack operation instructions

V. Handler Deformation Engine

It is obvious that the core of the deformation engine is the design of the template. The more templates are designed, the better deformation results of the handlers. However, in the process of the handler deformation iteration, we must guarantee the equivalent before and after deformation (including the same results after executing, the same influence on flag register). Deformation templates are designed in NISLVM by two principles:

- Decompose the stack operation instructions;
- Split some arithmetic and logic instructions based on discrete mathematics.

There are no templates that are designed to split the arithmetic and logic instructions based on discrete mathematics in the traditional VM-based software protection. Consequently, combined with the two principles, it will increase the difficulty when attackers analyzing the handlers.

The two type templates will be introduced in the following:
To design the template based on this principle we need to consider the influence on the flag registers. As shown in Fig.8, the “xor” instruction is deformed by two iterations.

\[
\begin{align*}
A \& B &= (A \& B) \&(C \mid C) = (A \& B) \&(C \mid C)
\end{align*}
\]

The “xor” instruction of the original handler before deformation may set the ZF to 1. Then after two iterative deformations, the “xor” instruction set ZF to 1. However, the added instruction “add” deformed from the “pop” instruction may make the value of the flag register incorrectly. In NISLVM, we solve the problem by saving and restoring mechanism to protect the flag register not to be affected when designing the deformation engine. As follows:

```plaintext
pop eax
push edx
xor edx, eax
mov edx, [esp+4]
xchg edx, dword ptr ss:[esp]
xchg edx, dword ptr ss:[esp+4]
xchg edx, DWORD PTR [esp]
xor DWORD PTR [esp], edx
push edx
pushfd
```

So the deformation engine of the NISLVM has accomplished to deform almost all instructions in handlers. It is obvious that, the more operations can be deformed, the less similarity of each handlers in the protected software. This reduces the possibility of “BORE” (Break Once, Run Everywhere).

2). **Split the arithmetic and logic instructions based on discrete mathematics**

Some operations can be split into other operations that are more complicated. According to following formulas:

\[
\begin{align*}
A \oplus B &= (A \& -B) \oplus (A \oplus B) \\
A \oplus B &= (A \oplus B)(C \oplus -C) = (A \oplus B)(C \oplus -C) \\
A \& B &= (A \& B)(C \& -C) = (A \& B)(C \& -C)
\end{align*}
\]

Of course, there may be more fomulas that can be used to split the arithmetic and logic instructions. Here, take the first fomular for example, we can get relative deform template as following:

```plaintext
add r1, r2
mov [esp-4], r1
xor [esp-4], r2
and r1, r2
shl r1, 1
add r1, [esp-4]
```

So above two rules can make the deformation of the handler more diverse. And then, it can improve the diversity and security of the protected software by NISLVM.

VI ANALYSIS AND EXPERIMENT

A. **Security and Performance analysis**

Generally speaking, it is the expense of running performance that usually acts as the negative affect caused by software protection, especially VM-based technology. In the following, we will analyze the security and performance influence of the software protected by NISLVM.

1). **Security analysis**

When attacker wants to analyze the algorithm of the protected instructions correctly, he must face following problems: ① Determine the right arrangement of the registers in VM_Context1 and VM_Context2 strictly before and after each RRI. The protected software may be contained much RRIs Obviously, the RRIs needs attacker spend a mass of energy and time. ② Obtain the registers’ value, and the value of registers should be calculated correctly according to correlative rules. Besides, it can increase more difficulty to get the vm_context by more complexity calculating rules between VM_Context1 and VM_Context2. ③ Analyze instructions in handlers and determine the function of the virtual instruction. However, the instructions in handlers have been deformed, and more deformation templates make them more diverse, implementing the same function with very different instructions. To some extent, this reduces the possibility of "BORE" (Break Once, Run Everywhere).

Moreover, ADIs can make the anti-debug mechanism more concealed and secure, and this technique make that: different execution environments may lead to different response measures, different response measures may lead to different execution results. That is to say, it increases the security to prevent being debugged with the random using of various interactive anti-debug mechanisms.

Beside, the NISLVM has the same protection effectiveness with traditional VM-based software protection.

2). **Performance analysis**

Now, we will take an example to analyze performance influence, and the original instruction is on the left, the virtual instruction is on the right.

- **ADD EAX, 8**

<table>
<thead>
<tr>
<th>Original instruction</th>
<th>Virtual instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD EAX, 8</td>
<td>move ADDR, 3h</td>
</tr>
<tr>
<td></td>
<td>load DWORD PTR [ADDR]</td>
</tr>
<tr>
<td></td>
<td>move ADDR, 2h</td>
</tr>
<tr>
<td></td>
<td>load DWORD PTR [ADDR]</td>
</tr>
<tr>
<td></td>
<td>xor DWORD</td>
</tr>
<tr>
<td></td>
<td>ri 1h, 5h</td>
</tr>
<tr>
<td></td>
<td>load DWORD 8h</td>
</tr>
<tr>
<td></td>
<td>add DWORD</td>
</tr>
<tr>
<td></td>
<td>move ADDR, 3h</td>
</tr>
<tr>
<td></td>
<td>load DWORD random</td>
</tr>
<tr>
<td></td>
<td>ri 0h, 2h</td>
</tr>
<tr>
<td></td>
<td>store DWORD PTR [ADDR]</td>
</tr>
<tr>
<td></td>
<td>xor DWORD</td>
</tr>
<tr>
<td></td>
<td>move ADDR, 2h</td>
</tr>
<tr>
<td></td>
<td>store DWORD PTR [ADDR]</td>
</tr>
</tbody>
</table>

Firstly, one original instruction “ADD EAX, 8” is interpreted by several virtual instructions, and generally, one virtual instruction will be interpreted by several handlers. Handlers cannot be given here for lack of space.
Secondly, according to the virtual instructions, we can see that “double VM_Context” brings more operations. At the beginning to use the registers, it needs to get values by computing the values in double VM_Context, which can be achieved by “load” and “xor” virtual instructions. And at the end to use, it needs to set the value in double VM_Context, which can be achieved by “store” and “xor” virtual instructions.

Thirdly, handlers will be deformed with the deformation templates, and users can set the deformation iteration according to their demand.

Then assume x, y and z respectively stand for double VM_Context technique, handler deformation technique and security virtual instruction technique. Performance “P” is the function of x, y, z. Look at the following formulas:

\[ P = f(x, y, z) + k \]

“k” includes other factors impacting “P”, such as added “jump”. “x” brings splitting and computing operations, handler deformation make handlers more complicated than original handlers and security virtual instructions need separate instructions to implement their functions. So all of them will influence “P”, but we can also know that: (1) even if all virtual instruction will use register value, additional instructions of operations are times of number of virtual instructions, so the relationship between “x” and “P” must be linear. Even the computing rules are more complicated. (2) The Influence of “y” to “P” depends on the iteration time, which users can control. The bigger the iteration time, the more influence on “P”, and the relationship between them is linear, too. (3) “z” stands for RRI and ADI, which also can be controlled. The influence to performance depends on the number of RRI and ADI used in protected software, but less influence than “x” and “y” generally. Therefore, in following experiment, we mainly verify the influence to “p” caused by “x” and “y”.

In fact, when protecting software, users often choose the best protection if the performance overload is in the acceptable range.

### B. Experiment

We have done some experiments to analyze change of protected software in file size (KB) and the influence to performance which embed through executing time (ms) of the software before and after protection.

1). **Experiment environment**
- CPU: 3.0GHz.
- Memory: 4G

2). **Experiment objects**
- Sorting algorithm: SelectSort, ShellSort, HeapSort and QuickSort

3). **Experiment process**
   a. Construct four sorting algorithms mentioned above, and use them to sort 200 inversion numbers. And then sort these numbers.

   b. Select the instructions to be protected in the four algorithms, furthermore, number of the instructions selected are the same when protecting the same algorithms.

   c. Determine the deformation iteration time which is 40 with NISLVMP technology.

   d. Obtain the file size and executing time of four unprotected sort algorithm. Then protect the four algorithms by NISLVMP, VMProtect and Code Virtualization. But NISLVMP protection is split into two modes: one uses the single VM_Context, and the other uses double VM_Context. In the experiment, we did not insert SVIs when protecting these algorithms. And also, the protection level of four protection technologies is medium. The executing time is obtained by the average time for ten executions.

4). **Experiment results and analysis**

**Table.3 and Table.4** show the results of file size and executing time separately.

<table>
<thead>
<tr>
<th>Algorithm Options</th>
<th>SelectSort</th>
<th>ShellSort</th>
<th>HeapSort</th>
<th>QuickSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected</td>
<td>528</td>
<td>528</td>
<td>528</td>
<td>528</td>
</tr>
<tr>
<td>NISLVMP with Single VM_Context</td>
<td>540</td>
<td>544</td>
<td>540</td>
<td>544</td>
</tr>
<tr>
<td>NISLVMP with Double VM_Context</td>
<td>542</td>
<td>546</td>
<td>542</td>
<td>546</td>
</tr>
<tr>
<td>VMProtect</td>
<td>552</td>
<td>552</td>
<td>552</td>
<td>552</td>
</tr>
<tr>
<td>Code Virtualization</td>
<td>549</td>
<td>549</td>
<td>549</td>
<td>548</td>
</tr>
</tbody>
</table>

As shown in the **table.3**, file size of protected algorithms change a little, because all the VM-based software protection will embed the interpreter into the software. The NISLVMP affect little on the file size perhaps because we do not design junk handlers in the interpreter, and perhaps the iteration times of handler deformation are different among them. The file size of software protected by NISLVMP with double VM_Context is a little bigger than protected by NISLVMP with single VM_Context because there are some special handlers to achieve setting and getting the register value of two VM_Context.

<table>
<thead>
<tr>
<th>Algorithm Options</th>
<th>SelectSort</th>
<th>ShellSort</th>
<th>HeapSort</th>
<th>QuickSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected</td>
<td>27.9</td>
<td>23.6</td>
<td>23.7</td>
<td>23.3</td>
</tr>
<tr>
<td>NISLVMP with Single VM_Context</td>
<td>375</td>
<td>81.2</td>
<td>475</td>
<td>293.7</td>
</tr>
<tr>
<td>NISLVMP with Double VM_Context</td>
<td>678</td>
<td>141</td>
<td>756</td>
<td>495</td>
</tr>
<tr>
<td>VMProtect</td>
<td>168.6</td>
<td>42</td>
<td>182.7</td>
<td>128.1</td>
</tr>
<tr>
<td>Code Virtualization</td>
<td>464</td>
<td>87.8</td>
<td>578</td>
<td>306.1</td>
</tr>
</tbody>
</table>

As shown in the **table.4**, executing time of algorithm protected by VMProtect is the least, but the executing time of the same protected algorithms have the same order of
magnitude. The executing time of “ShellSort” algorithm is similar with other algorithms before protected, but it cost less than other algorithms after protected, the reason is that there are more compare operations and less swap operations. In a word, the new improvements in NISLVM do not affect the performance much more than other VM-based software protection.

VII CONCLUSION AND NEXT PLAN

In this paper we propose a VM-based software protection called NISLVM, which has been improved in three aspects: double VM_Context, security virtual instructions and more deformation template in deformation engine. NISLVM not only has the protection strength of the traditional VM-software protection, but also preliminary efficiency analysis showed that the improvements increase security of software against adversaries to some extent, and influence on performance can be adjusted according users’ demand. Of course, the improvements need to be improved continually. That is to say, we will go on to research on how to increase software’s security but affecting performance least. Further more, how to evaluate the effectiveness of the VM-based software protection exactly is also an important problems.

REFERENCES