SymSem: Symbolic Execution with Time Stamps for Deobfuscation

Huayi Li[(✉)], Yuanyuan Zhan, Wang Jianqiang, and Dawu Gu

Shanghai Jiao Tong University, Shanghai, China
lihuayi_sjtu@outlook.com, zhang-yy@cs.sjtu.edu.cn, wjq.sec@gmail.com, dwgu@sjtu.edu.cn

Abstract. Code virtualization technique obfuscates programs by transforming original code to self-defined bytecode in a different instruction architecture. It is widely used in obfuscating malware for its ability to render normal analysis ineffective. Using symbolic execution to assist in deobfuscating such programs turned to be a trend in recent research. However, we found many challenges that may lead to semantic confusion in previous symbolic execution technique, and proposed a novel symbolic execution technique enhanced by time stamps to tackle these issues. For evaluation, we implemented it as a prototype of SymSem and deobfuscated programs protected by popular virtual machines. The results indicate that our method is able to accurately recover the semantics of obfuscated function trace.

Keywords: Deobfuscation · Virtualization obfuscation · Symbolic execution · Trace rewriting

1 Introduction

Code protection techniques help software writers protect their copyright, these techniques also become weapons against malware analysis. Popular code protection techniques includes control flow flattering, junk code, string encryption and code virtualization. Among all the developed code protection techniques, code virtualization, also known as VM-based code obfuscation is one of the most practical and effective code obfuscation techniques for its ability to defeat unauthorized code analysis in either static or dynamic manners. It is also empowered by combining itself with other techniques.

The key idea of VM-based obfuscation is transforming the instruction set of the original program to another one with semantic invariance and embedding the obfuscated program with an emulator to execute the generated code. While each VM-based code obfuscator realizes its own instruction set and emulator, modern virtual machines still can be divided into two main types of realizations.

One uses a dispatcher-handler model, whose emulator can be clearly divided into two components. The main component, dispatcher, reads one instruction from the code area and decodes its operation type. Then it picks up a corresponding handler to perform the code-specific operation, which may add two integers or perform memory operation. For example, *VMProtect 3.09* [4] applies its emulator with a few dispatchers with handler tables, each of them contains pointers of hundreds of handlers.

The other main type of virtual machines apply the direct-threading model. These virtual machines eliminate the dispatcher, which is vulnerable in reverse engineer. Instead, these virtual machines enhance the original handlers by append decoding functionality at the end, whose logic can be costume designed, thus enabling the virtual machine to further obfuscate its control flow features. Recently, many commercial code obfuscators such as *VMProtect* [4], *Themida* [3], *Code Virtualizer* [1] have all adapted direct threading model.

The demands to analyze the VM-based code obfuscation increase due to its growing popularity in malware protection. To deobfuscate malware protected by code virtualization technologies, researchers have developed many techniques aiming for automatic analysis. For example, VMAttack [11] detects the dispatcher-handler loop and uses folding optimization to recognize important instructions in the virtual machine code. VMHunt [18] applies data flow analysis on the trace of obfuscated program and identify the entries and exits in the trace, then it uses symbolic execution technologies to generate a formula to represent the trace's semantics. Liang [12] developed a method of trace simplification based symbolic execution and compilation optimization. These works all based themselves on dynamic analysis, including data flow analysis, taint analysis and symbolic execution.

However, we found that former works used a simple model when applying symbolic execution, this model may have a few inputs, but with only one output. While practical code in virtual machines comply with a different model considering complicated semantic meanings. First, a snippet of virtual machine code may have multiple inputs and outputs. These outputs are physically separated thus can not be represented in one expression. For example, a handler may read a bundle of inputs from memory, do some calculation and save the results to different registers and memory addresses. What's more, many complex situations which would cause semantic confusions have been omitted. A handler with two memory inputs may have two symbolic values both point to the same address under specific conditions. Improper treatment of these two values may cause serious semantic confusions for analyst and lead to different semantic comprehension in recovering code.

We found that, besides explicit dependencies, instructions and its generated values have implicit relationship that make a difference to the output semantics in symbolic execution. Any misconduct in symbolic execution may lead to loss of information needed for resolving semantic confusions in code recovery. As these implicit relationships are conducted under time sequences, we further noticed that time stamps is an important indicator of the relationship.

In this paper, we systematically study the challenges and develop a new approach in symbolic executions and code recovery. First, we elaborate the challenges when applying a more practical semantic model in symbolic execution. These challenges we found are caused by misconducts when dealing with multiple symbolic expressions. Then, we propose a new kind of symbolic execution technique enhanced with time stamp, which efficiently complements the missing information in original symbolic execution methods and help tackle the challenges we found. Finally, we implement a prototype of our symbolic execution method, SymSem. SymSem takes an obfuscated program trace as input, extracts the virtual machine code from the trace and rewrites these code by symbolic execution and recovering code from symbolic expressions. We will illustrate how our implementations can be used in trace optimization and reverse engineer.

For evaluation, we implement a prototype of SymSem based on symbolic execution with time stamps. We pick up test cases from famous algorithms' realizations and open-source projects, including *binary search*, *matrix multiply*, *tcp_checksum*, *rc4 encryption* and *bzip2.14*. We obfuscate these programs with commercial code virtualization obfuscators like *VMProtect* and *Themida*. Then, we trace the obfuscated program and rewrite the obfuscated function. The rewritten traces have the same semantic meaning when executing in the obfuscated program. Moreover, the trace length is reduced to 12.5% to 36.68% of the original. Our evaluation also shows that SymSem can be easily scaled to large-size traces for two reasons. First, the time complexity of our prototype is more related with the handler numbers and their length, not the trace length of obfuscated function. Also, SymSem are designed for parallelism as symbolic execution procedures of different handlers can be done concurrently.

This paper makes the following contributions.

1. We found a series of problems related to trace rewriting by means of symbolic execution in practical situations.
2. We designed a new symbolic execution method to precisely and accurately extract the semantics of obfuscated program. Our symbolic execution with time stamps is able to tackle the difficulties related to extracting semantics through symbolic execution.
3. Based on our new symbolic execution method, we designed a prototype of SymSem and we evaluate it in different commercial virtual machines. The result shows that it is able to rewrite the trace of obfuscated function.

The rest of the paper is organized as follows. Section 2 points out the limitations in symbolic execution and the challenges for this work. Sections 3 and 4 describes the design of SymSem. The implementation and performance evaluation on SymSem is in Sects. 5 and 6, respectively. Section 7 lists the related works. Sections 8 and 9 discuss the limitation in our work and conclude the paper.

2 Challenges

Symbolic execution has been used in deobfucating in many previous work due to its ability of forming corresponding relationship between inputs and outputs. The results of symbolic execution, a set of symbolic expressions,which represents the relationship between program inputs and outputs, have been not only the final output of deobfuscation analysis [18], but also compiled to optimized code with better readability [12]. Unfortunately, we first found previous work made an impractical assumption that target programs with only one output, thus only one symbolic expression was needed to represent the semantics of target code. This assumption violates the model of practical programs, as a program may have multiple separate input sources and outputs.

We then found that current form of symbolic expressions are unable to contain the full semantic information of program trace thus may cause semantic confusions. Meanwhile, generating compilable code from multiple symbolic expressions have many challenges with explicit and implicit data dependence.

In this chapter, we will describe the challenges we encountered when rewriting program trace with multiple outputs. These challenges are omitted by previous work. Then we will unveil why current symbolic expressions may cause confusions.

2.1 Challenges with Multiple Symbolic Expressions

We use the example below to illustrate one of the challenges when representing the semantics of program trace using current symbolic expression.

```
mov ecx, dword ptr [eax+4]
add dword ptr [eax+4], 4
```

Symbolic executing two lines of code above will generate two expressions below.

$$\begin{aligned} &ecx: \ dword\ ptr\ [eax+4] \\ &dword\ ptr\ [eax+4]: \ dword\ ptr\ [eax+4]+4 \end{aligned}$$

It is obvious that the value in register *ecx* relies on the value saved in $[eax+4]$, thus a read after write hazard may arise when rewriting procedure fails to preserve the appropriate sequence of code generation based on symbolic expressions. In this case, read from dword address $[eax+4]$ must precede the writing operation of the same address. Read after write may lead to different semantic meanings.

Usually, the first idea of solving this hazard follows the method of constructing a graph of dependence. On this graph, the code generation for expression *ecx* relies on the read operation of the initial value of address $[eax+4]$.

2.2 Challenges with Alias Symbolic Values

In addition to the hazards above, there is another challenge which may cause semantic confusion when using previous symbolic execution method. We call it alias, which results from the inability of judging the equality of two symbolic values in outcome expressions.

```
mov dword ptr [eax+4], 0xbeefdead
mov dword ptr [ebx+4], 0xdeadbeef
mov ecx, dword ptr [edx]
```

Here we illustrate the alias problem by the code above. We can easily enumerate the symbolic expressions generated as below.

$$ecx: \ dword\ ptr\ [edx]$$
$$dword\ ptr\ [eax+4]: \ \texttt{0x}beefdead$$
$$dword\ ptr\ [ebx+4]: \ \texttt{0x}deadbeef$$

Based on these expressions, we are unable to judge that if *edx* equals to *eax*+4, neither the case if *eax* equals to *ebx*. When *edx* equals to *eax*+4, the first expression could also be rewrited as $ecx: \texttt{0x}beefdead$. Meanwhile, the rewriting procedure may decide to first deal with the first expression. The generated code would simply move the initial value stored in address *edx* to register *ecx*, which is contrary to the true sequence in the original code. This results in a new kind of hazard that we call write after write hazard and the situation also exists when *edx* equals to *eax*.

The case above shows that different sequences of code generation for expressions with possible alias value may generate code with different semantic meanings. In addition, the problem could not be solved by dependence graphs as they are limited by implicit dependence relationship. Meanwhile, these relationships, which are originally organized by time sequence, have been eliminated in the symbolic expressions above. For those who want to understand the state of program by these expressions, confusions are inevitable.

3 Symbolic Execution with Time Stamps

As previous form of symbolic expressions are unable to contain all information necessary for generating compilable code. We implemented a new kind of symbolic execution method named symbolic execution with time stamps. As is indicated by its name, symbolic execution with time stamps tags every symbol value with a time stamp when they are created. These time stamps demonstrate the sequence of IR generation when needed. Here we describe how time stamps help symbolic execution tackle the challenges referred in Sect. 2. In Sect. 4.3 we will further elaborate our methods by an detailed example.

3.1 How Time Stamp Tackles Read After Write Hazards

First we describe how time stamps help tackle the challenge of read after write hazards. Though this challenge can also be tackled by a dependence graph, time stamp is a more natural method with generality.

We take the first case in the Sect. 2.1 as an example. The code first saves the value of address [*eax*+4] to *ecx*. Then it updates value of address [*eax*+4] with a new value. Under symbolic execution with time stamps, the value in the register *ecx* will be tagged with time stamp 0, as it is born in the operation of the first instruction. Similarly, the newly updated value of address [*eax*+4] will be tagged with time stamp 1.

When it comes to code generation, the procedure will first deal with values tagged with lower time stamps. In this case it corresponds to the symbolic value in *ecx*. It will read a value from address [*eax*+4]. Here of course the read operation will get the right value, instead of one has been updated. Then the procedure deals with the value tagged with time stamp 1. The problem of read after write hazards is solved naturally.

3.2 How Time Stamp Tackles Alias Values

Time stamps can also help tackle the challenge of alias values, which is invisible in dependence graphs. Still, we take the case in Sect. 2.2 as an example. It is multifarious to enumerate all possible alias cases in the example. Not to mention generating conditions and constraints of each alias cases.

However, under symbolic execution with time stamps, the value in address [*eax*+4], [*ebx*+4] and *ecx* is respectively tagged with time stamp 0,1 and 2. In the following IR generation procedure, if we follow the sequence of time stamps, the value in the address [*eax*+4] will be first updated. Then it comes to the value in address [*ebx*+4] and *ecx*. Never mind what alias cases may happen, the IR generation procedure follows the correct sequence and avoids any conflicts. So we can apply the same solution to tackle the alias challenge.

4 Design

4.1 Overview

We present a new method of symbolic execution and implemented it in a new program trace analysis system named SymSem, which is used to analyze traces of programs obfuscated by code virtualization technique. In this chapter we will illustrate the overview of SymSem.

SymSem is a system aiming to recover the semantics of the trace obfuscated by code virtualization technique. It takes the trace as input and outputs the LLVM IR of the trace, which is both readable and compilable. Finally, SymSem compiles the IR to generate assembly code for further evaluation. Figure 1 depicts the whole architecture of SymSem, including three kernel components which would be described in detail in the following sections.

1. VM Architecture analysis. For a program protected by code virtualization, we first run it and record the trace. We implemented an analysis method based on execution rate to extract virtual machine entries, exits and all handlers in the trace.

2. Symbolic execution with time stamps. We assume that the semantic meanings of the program can be represented by a set of expressions between inputs and outputs. Based on this assumption, the semantic meaning of obfuscated function can be achieved by connecting all the semantic expressions of handlers in the trace. Here we use a new kind of symbolic execution technology named symbolic execution with time stamps to extract semantic representation of all handlers, VM entries and exits in the trace.
3. Generating LLVM IR. SymSem generates IR for LLVM compiler based on the results of symbolic execution. These IR can be used to generate code or symbolic representation of the whole trace.

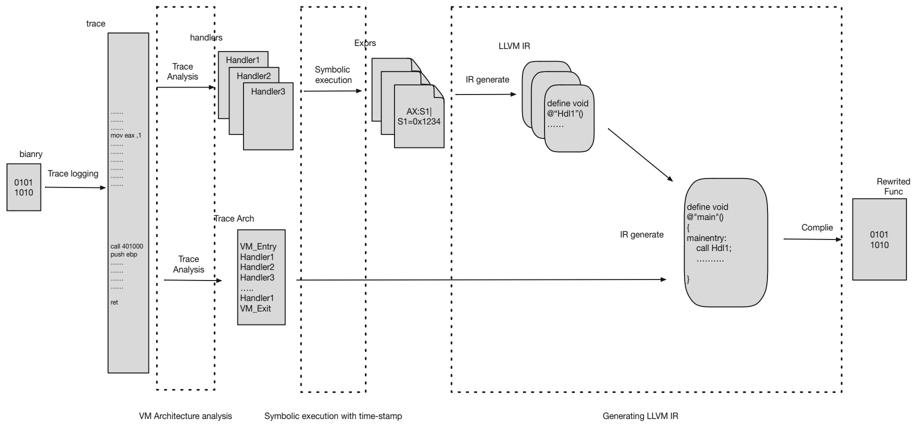


Fig. 1. SymSem architecture

4.2 VM Architecture Analysis

As described in Sect. 4.1, the VM architecture analysis is designed for extracting virtual machine code from program trace. These code contain handlers and information about how to reorganize the generated intermediate representation.

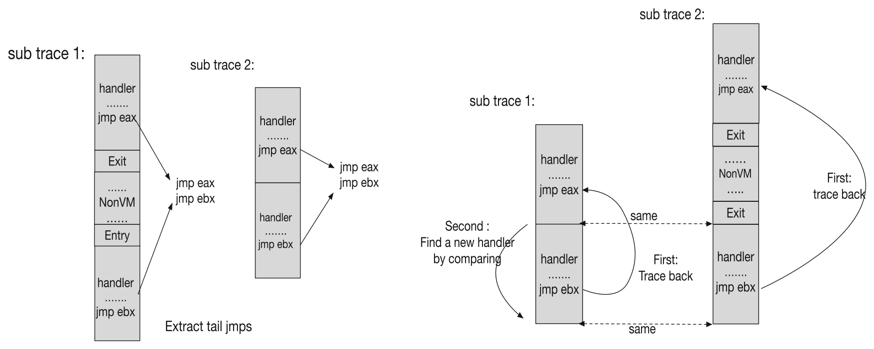
Here we use an easily implemented method to analyze the program trace. We also assume that there does exist at least one virtual machine in the trace. Our methods is based on two phenomenons we observed below:

1. For a specific program protected by code virtualization, all virtual machine handlers have the same type tail jump, such as "*jmp* 0x434343" or "*jmp register*".
2. In the program trace, those basic blocks which are not in any loop but belong to handlers have more chance to be executed for multiple times when considering a mount of continues instructions.

Method Overview. Our method can be simply described as two steps. First, screen out possible basic blocks belong to handlers based on execution rate. Then, make further analysis based on control flow and data flow. These analyses will finally help us mark all the basic blocks as "VM code" or "Non-VM code".

Method Details. We will describe the details of our method in this section. SymSem first runs the program and trace all the executed instructions once. Also, for every basic blocks to be executed, SymSem records the address of its first instruction and the value of stack register. These two kinds of records are saved separately in two files.

We use the first trace file, which contains traces of all instructions, to calculate the occurrence frequencies of different unconditional jump instructions. Like what is shown in Fig. 2(a), we rank these instructions and assume those with higher occurrence frequency are candidates of tail jump instructions of handlers.



(a) extract tail jmps (b) detect handlers

Fig. 2. Extract tail jmps & detect handlers

Further analysis was based on control flow. We assume that the code before a handler either belongs to another handler or the VM entry. Similarly, the code after a handler is another handler or the VM exit. As is shown in Fig. 2(b), for every jump instructions we have screened out, we trace backwards to another jump instruction. The code between the two tagged jump instructions must be a handler or includes one exit, one entry and some Non-VM code. We use some data flow features to distinguish between this two situations. In the first case, those code between two tagged jump instructions are tagged as a handler.

Finally, SymSem runs data flow analysis to recognize the entries and exits of virtual machine. The VM entries and exits have some specific data flow features which are easily recognizable. One of them is that all VM entries save the general registers to some region of the memory. In this way we distinguish the code between entry and exit as VM code and the other as Non-VM code.

4.3 Symbolic Execution with Time Stamps

The components of symbolic execution and IR generating is the key of SymSem. We use symbolic execution technique we first presented in Sect. 3. In this section we elaborate our symbolic execution method in detail by an example.

A Detailed Example to Illustrate Our Method. Here we use an example to elaborate how symbolic execution works on actual code. The case here is a simplification of code we actually encountered.

```
mov ecx, dword ptr [eax+0xc]
add dword ptr [eax+0xc], 0x4
mov dword ptr [ebx+0xc], 0xdeadbeef
mov edx, dword ptr [edx]
```

This code fragment first reads a symbolic value from address [*eax*+0xc] to *ecx*. Then it adds the value in memory address *eax*+0xc] with 4, writes a concrete value to address [*ebx*+0xc]. Finally, it reads a symbolic value from address [*edx*] to register *edx*.

It is clear that *eax* and *ebx* can be two alias values and there is no chance to eliminate the alias simply based on the code fragment. Under that condition, the address [*eax*+0xc] and [*ebx*+0xc] may have a write after write hazard. Also, there is a read after write hazard with address [*eax*+0xc]. The generated code must first read the value in memory address [*eax*+0xc], then write a new value to the same address. What makes things more difficult is that *edx* may equals to *eax*+0xc or *ebx*+0xc, making the situation more complicated.

Symbol Definition. We use the definitions below to elaborate the symbols in the following.

Definition 1. There are two types of value in symbolic execution, concrete value and symbolic value. We use C_n^m S_n^m to represent them. Here, m represents the width of the value. And, n is a unique id of every value. For example, S_0^{32} represents a symbolic value of 32 bit size and its id is 0.

Definition 2. We use symbol | to represent the bind of values. For example, $C_1^8|_{C_1^8=\text{0x12}}$ represents a concrete value of 0x12, while $S_1^{32}|_{S_1^{32}=S_0^{32}}$represents a symbolic value equal to S_0^{32}.

Definition 3. We use symbol $T_V = t$ to indicate the time stamp of value V. The time stamp of $[V_1, V_2, \ldots, V_n]$ is the newest one of $V_1, V_2, \ldots, V_n$. for simplicity, time stamps start with 0.

Definition 4. We use a set of expressions like $Pos_1 : V_1, Pos_2 : V_2, ...$ to represent the state of system. $EAX : C_0^{32}|_{C_0^{32}=\text{0x12345678}}$ means the value in register *eax* is 0x12345678.

Procedure of Symbolic Execution with Time Stamps. We assume the initial state of the system can be represented as $mem[eax{+}\texttt{0x}c] : S_0^{32}, mem[ebx{+}\texttt{0x}c] : S_1^{32}, mem[edx] : S_2^{32}$

Now we elaborate the procedure of symbolic execution with time stamps. After the symbolic execution of the first instruction, register *ecx* was written with a new symbol value, which equals to the value saved in memory address [*eax*+0xc]. As it was the first instruction, the new created value will be tagged with time-stamp 0.

$$ecx : S_3^{32}|_{S_3^{32}=S_0^{32}, T_{S_3^{32}}=0}$$

After the second instruction was executed, the value in the memory address [*eax*+0xc] have been updated, and the time stamp of the new value is 1.

$$mem[eax + \texttt{0x}c] : S_4^{32}|_{S_4^{32}=S_0^{32}+4, T_{S_4^{32}}=1}$$

The third instruction sets address [*ebx*+0xc] with a concrete value 0xdeadbeef. Its time stamps is of course 2.

$$mem[ebx + \texttt{0x}c] : C_0^{32}|_{C_0^{32}=\texttt{0x}deadbeef, T_{C_0^{32}=2}}$$

When the final instruction is executed, register *edx* will be updated with a new value from address [*edx*]. This operation creates implicit data dependencies with *edx* and preceding address values. Here we can enumerate the final state of the symbolic execution.

$$\begin{aligned}
&ecx : S_3^{32}|_{S_3^{32}=S_0^{32}, T_{S_3^{32}}=0}\\
&mem[eax + \texttt{0x}c] : S_4^{32}|_{S_4^{32}=S_0^{32}+4, T_{S_4^{32}}=1}\\
&mem[ebx + \texttt{0x}c] : C_0^{32}|_{C_0^{32}=\texttt{0x}deadbeef, T_{C_0^{32}=2}}\\
&mem[edx] : S_5^{32}|_{S_5^{32}=S_2^{32}, T_{S_5^{32}}=3}
\end{aligned}$$

If we observe the final state of the code, we will find that all values have been tagged with a time stamp after symbolic execution. When generating corresponding operation in intermediate representations, we can avoid the confusion of alias as well as the read after write hazard or write after write hazard by following the sequence of time stamps.

4.4 IR Generation and Compilation

The method of IR generation includes two steps. The first step is generating IR for each handler extracted from the trace. All these semantic representation have multiple inputs and outputs. Then these IR will be concatenated to form the semantic representation of the whole trace.

Algorithm 1. IR generate algorithm for handlers

```
Input: Expression_set $\Phi$
Output: LLVM IR of Code
Function Init_Dependence_Graph (Expression_set $\Phi$)
    Dependence_graph G ← $\emptyset$
    for each Expression E in Expression_set do
        for each Expression $E_S$ in SubExpression($E$) do
            add edge ($E_S \leftarrow E$) to G
        if value E needs to be written to address addr then
            add edge (addr ← $E$) to G
    return G
Function Generate IR(Expression_set $\Phi$)
    Dependence_graph G ← Init_Dependence_Graph($\Phi$)
    Q ← empty Queue
    for each timestamp $t \in Timeline$ do
        Expression_set $\theta \leftarrow \bigcup$ expressions E with $T_E = t$
        if $\theta = \emptyset$ then
            continue
        for each Expression $E \in \theta$ do
            r ← BuildIR($E$)
            Remove edges in G whose source is E
            if value E needs to be written to address addr then
                append (r,E,addr) to Q
        Loop ← True
        while Loop do
            Loop ← False
            for each IR r,Expression E,address addr in Q do
                if there is no edge point to E in Graph G then
                    Build_IR_For_Store (r,addr)
                    Loop ← True
                    remove(r,E,addr) in Q
```

Generate IR for Handlers. SymSem uses LLVM APIs to build two basic function for IR generation, BuildIR and Build_IR_For_Store. The former takes a symbolic expression, read necessary data from memory and registers, do the arithmetic operation required to generate corresponding representation. While, Build_IR_For_Store takes output of the former function and writes the result to target register of memory.

SymSem utilizes the two functions above and time-stamp information provided by previous phases to generate IR for handlers. It invokes the BuildIR function under the sequence of time-stamps. The generated representations will be put in a queue, waiting for writing to target until conditions satisfied.

In consideration of saving physical registers and memory space, SymSem uses a dependence graph here instead of time stamps. This also accelerates the writing procedure. The algorithm used to generator LLVM IR correspond to handlers can be described in Algorithm 1.

Here we take the code in Sect. 4.3 to illustrate how our algorithm works. The first step of the algorithm is to generate representation for S_3^{32}, as it is the only one which is created at time zero. The procedure will read a double word value from memory [eax+0xc]. Then the algorithm deals with S_4^{32}. The value will be immediately written to memory as there is no value explicitly dependent on it. Next, the same memory update can be applied to address [ebx+0xc] for the same reason. Finally, the value S_5^{32} can be written to register *edx*.

Generate IR for Virtual Machine Trace. Having generated the IR of different handlers, the key of further generating accurate IR of the obfuscated program trace is to efficiently organize the IR of handlers.Here we define semantics of each handler as a function. Then the semantics of virtual machine trace can be represented by a sequence of function invokes. As the semantic representation is in the form of LLVM IR, the result can be conveniently compiled into executable code.

5 Implementation

We realized a prototype named SymSem for our methods and use a custom PIN tool which includes 162 lines of C code as trace recorder. The symbolic execution engine is based on manticore [2], most of the changes we made on it is meant to add time-stamp in its CPU emulation module and to analyze the result of symbolic execution. Our tool contains 25009 lines of code, 8545 lines of which are newly added to the original framework, including 2237 lines related to symbolic execution.

The PIN tool used in evaluation has about 225 lines of C code. Its responsibility is to load the rewritten code into memory and execute.

An interesting result we discovered is that not all LLVM passes could be implemented on our intermediate representation. So we have to use *r0* optimization of opt tools with selected passes including *"-reassociate"*, *"-adce"*, *"-mem2reg"* which will not lead to semantic error. We found most errors those unfit passes incurred is due to they made a false assumption of stack usage of our code.

6 Evaluation

In this chapter we will evaluate our method on different code virtualization protection technologies. Our evaluation focus on three questions. (1) Can SymSem correctly reverse engineer the architecture of the virtual machine from the obfuscated trace? (2) Is SymSem able to recover the semantics of obfuscated program trace accurately? (3) How long does SymSem cost and how much can parallelism help the analysis?

6.1 Experimental Framework

We set up our experiment on a Windows XP virtual machine. All the tested programs were compiled by visual studio 2008 on the same machine. The remaining analysis were conducted on a server with Intel Xeon Gold 5122 and 128G of RAM, which runs Ubuntu 18.04.

We choose our test programs based on five algorithms including *binary serach*, *matrix multiply*, *tcp_checksum*, *rc4 encryption* and *bzip2 compression*. We choose proper implementations for these algorithms from Github and other open sources. Two famous commercial code virtualizer *VMProtect 3.09* and *Themida 2.4.2* are used to obfuscate these programs. As required by the code virtualizers, we only obfuscate the function of chosen algorithms and exclude other code. For example, when dealing with the program of *rc4 encryption*, we only obfuscated the encryption function.

For all tested programs, we separately constructed inputs to make sure that they call the obfuscated function and exit normally. For example, the input of *rc4 encryption* is a private key file and a message file. We use PIN tools to trace the obfuscated programs. The tracer records not only instructions, but also their addresses and specific register values. We only trace the code which belong to the program itself and exclude the third party library. The following Table 1 shows statistics of the tested program traces.

As is show in Table 1, we count the trace length by its lines, the average length of handlers in the trace and if the data in virtual machine is encrypted.

Table 1. Tested programs. This table shows the characters of the tested programs. "Handler avglen" means the average length of handlers appeared in the trace. We set the label "encrypted" true if the data computing in the trace is encrypted.

	Name	Length of trace	Length of trace (obfuscated)	Handler avglen	Encrypted
VMProtect	binary search	31	18328	17.45	False
	matrix multiply	151	60105	17.78	False
	tcp_checksum	174	75041	19.25	False
	rc4	1484	395214	16.82	False
	bzip2	244345	292161	19.61	False
Themida	binary search	31	121417	454.33	True
	matrix multiply	151	166234	384.69	True
	tcp_checksum	174	249736	516.98	True
	rc4	1484	1175319	453.65	True
	bzip2	244345	986447	271.59	True

As referred by previous work, modern commercial virtual machines all apply redundant handlers and the state of the art also encrypted its computing data. In our test experiments, *VMProtect* and *Themida* apply two different design philosophy. The former implements RISC-like ISA while the latter is CISC-like.

After being obfuscated by code virtualization, the function trace also expands to hundreds of instructions.

6.2 Our Tool Can Accurately Analyse the Arch of VM Trace

In this section we evaluate SymSem by its ability to recognize VM code. First, we use SymSem to find out handlers appeared in the trace, we compared them with the results of manual work. Then we use our tool to find out possible VM entries and exits, which were also compared with those we manually found.

SymSem tags basic blocks with three types, including "handler", "dispatcher" and "Non-VM". The former two types belong to virtual machines while the latter is not our rewriting target. We recognize VM entries and exits as special handlers. As one handler may have different control flow which leads to different semantic meaning, our tool specifically identified these cases.

Table 2. Handlers entries and exits found. This table lists the number of handlers found in the trace by manual work and automatic analysis. The word "hdl" is the shortcut of handler. Here "count once" means that different handlers with the same start address only count once. "(CF $\geq$ 1)" indicates that handlers have more than one control flow cases. The meaning of "(CF $\geq$ 2)" are similar.

	Name	hdl (manually)	hdl (count once)	hdl (CF $\geq$ 1)	(CF $>$ 2)	entry & exit
VMProtect	binary search	43	43	43	0	1
	matrix multiply	50	50	50	0	1
	tcp_checksum	52	52	52	0	1
	rc4	67	67	67	0	1
	bzip2	111	111	110	1	20
Themida	binary search	109	109	156	30	3
	matrix multiply	98	98	137	25	1
	tcp_checksum	117	117	181	39	3
	rc4	113	113	158	36	1
	bzip2	150	150	255	46	25

As is shown in Table 2, the handler number is at most 150 in the trace and SymSem correctly identified all handlers in the trace. It also identified many handlers which have same start address but different control flows. Programs of *bzip2* and *binary search* were found to have a few virtual machines inside the obfuscated function. The analysis output shows that between two virtual machines are some other function calls which are not obfuscated. SymSem does not target these code so it is necessary to recognize them.

6.3 Our Tool Can Accurately Recover the IR of vm Trace

The preceding analysis generates LLVM IR representation of the obfuscated trace. These IR representation can be compiled to get the optimized trace. As a correct optimized trace indicates the correctness of the generated IR representation, we use a custom PIN tool to test the correctness of the rewritten semantics.

This PIN tool writes the optimized trace into a unique region of memory and modify the PC register when the program is going to execute the origin code.

As our rewriting procedure is based on traces, test inputs must lead to the same control flow with those in the preceding analysis. We make sure that the test inputs lead to the same control flow with the trace file by limiting their ranges. For example, we use files with the same length in *tcp_checksum* and the same key in *rc4 encryption*.

We use a simple fuzzer which generates inputs with the same control flow to test the rewritten code. These inputs were also sent to the same program not instrumented. We compared all the outputs and found that the instrumented programs have the same outputs with those not instrumented. We treat this as a proof of the semantic invariance of rewriting procedure. From tests above, we made a conclusion that SymSem can generate rewritten semantics accurately.

After evaluating the semantic invariance of rewriting procedure, we also count the length of the rewritten trace, the length of IR and how many functions in it. Table 3 shows the statistics of the analysis result. We counted IR length by its number of lines, the most important data is the reduction rate which points out how much optimization SymSem made on the trace.

Table 3. Statistics of the analysis results. This table shows the statistics of the analysis results, which including IR representation and optimized trace. The reduction rate is calculated by comparing length of the trace in virtual machines with the length of optimized trace.

	Name	IR length	func in IR	Optimized trace	Trace in VM	Reduction
VMProtect	binary search	2936	143	5254	18328	28.66%
	matrix multiply	4231	197	21580	60105	35.90%
	tcp_checksum	4534	216	27532	75041	36.68%
	rc4	8649	175	110049	395214	27.84%
	bzip2	151456	6195	25248	83051	30.40%
Themida	binary search	19930	191	17496	119794	14.60%
	matrix multiply	17951	180	24886	166234	14.97%
	tcp_checksum	22957	219	34151	247834	13.77%
	rc4	20306	181	163747	1175868	13.92 %
	bzip2	804290	8280	105776	756225	13.98%

In the table, complicated programs such as *rc4* have longer trace, generated IR and more functions in the IR files. The results also show that the reduction rate is stable for specific obfuscators. The average reduction rate of *VMProtect* is 31.89% and the one of *Themida* is 14.24%. The gap indicates that the reduction rate is highly dependent on the obfuscator itself.

6.4 The Overhead of Our Tool

Based on the architecture of SymSem, the whole analysis time can be divided into three parts, the time of trace analysis, the time of symbolic execution and the time used to generate IR and assembly code. In this section, we hope to demonstrate that SymSem can analyze a program trace in an acceptable time even for a more complicated program.

The first part of analysis is the trace analysis. Our algorithm parses the whole trace twice. The first parse aims to get statistics of the basic blocks and control flow information. Then the second is able to recognize the VM code, including handlers, entries and exits. As this part of analysis time is heavily dependent on the length of trace, what makes a difference is the analysis time of symbolic execution and IR generation.

Here we separately tested the overhead of SymSem to generate rewritten assembly code. The tests below was based on single process, which means it can be further accelerated.

Table 4. Analysis overhead of single process. This table illustrates the overhead of symbolic execution and generation of the optimized assembly code. Tags are defined as follows: total_time = the total time of symbolic execution and generation of optimized trace, z3 = the seconds spent in ze solver, SE = symbolic execution, llc = the time spent in LLVM compiler, executed loc = the lines of code which is symbolic executed during the analysis.

	Name	total_time	z3	SE	llc	executed loc
VMProtect	binary search	1m14.982s	1.548s	1m14.022s	0.960s	733
	matrix multiply	2m2.686s	3.350s	1m53.162s	9.524s	889
	tcp_checksum	2m6.014s	3.297s	1m50.576s	15.438s	1001
	rc4	12m42.455s	3.315s	2m17.785s	624.670s	1127
	bzip2	4m58.226s	11.844s	4m44.088s	14.138s	2069
Themida	binary search	29m4.721s	163.150s	29m0.44s	4.278s	75418
	matrix multiply	11m56.508s	118.164s	11m52.812s	3.696s	56549
	tcp_checksum	38m5.542s	205.830s	37m58.790s	6.752s	98744
	rc4	32m43.882s	225.397s	29m49.469s	174.413s	72584
	bzip2	66min36.38s	283.747s	64m56.50s	99.878s	103996

Table 4 shows the analysis time, including the symbolic execution and IR generation. The analysis spends most of the time on symbolic execution. We study the relationship between the overhead of symbolic execution and the executed code length. The consuming time seems to be linear correlated with the executed length, excluding the case of *bzip2*. We guess the case of *bzip2* is an exception because it has more virtual machines than others to analyze, which cost a longer time.

The other part of analysis time, which is spent on compilers to generate assembly code is related to the length of obfuscated trace. The case of *rc4* has a great amount of encryption loop in the trace so it cost much more time to compile. However, of these ten different test programs, the longest analysis time is limited in 66 min. Considering the length of trace and the procedure can be further accelerated, we think this time is acceptable.

Furthermore, we tested how much can parallelism help in analysis. As described in, the analysis time can be divided into 3 parts and the second part, which generate LLVM IR for handlers through symbolic execution can be fully paralleled. We tested the overhead on programs obfuscated by *Themida*.

The symbolic execution can be accelerated by parallelism. Here we use multiple process to accelerate the analysis. Each process separately analyzes one handler concurrently. Then the symbolic execution result is assembled to generate IR representation and assembly code. We record the total time of generating executable assembly code. Table 5 reveals the whole test results.

Table 5. Multiple process analysis overhead. This table illustrates the total analysis time with multiple processes.

	Name	1 process	2 processes	4 processes	8 processes
Themida	binary search	23m29.353s	22m59.659s	12m0.215s	7m35.043s
	matrix multiply	11m56.508s	36min54.790s	19m4.849s	11m28.323s
	tcp_checksum	29m36.273s	25m16.340s	13m4.776s	8m10.550s
	rc4	32m43.882s	32m44.630s	24m49.204s	14m28.977s
	bzip2	37min16.617s	34min15.423s	17m37.760s	11m7.583s

For simplicity, we limit the process number by exponent of two. We set the upper bound as 8 because of physical resources limitations. In addition, for those test cases with multiple virtual machines in the obfuscated function, we only count the first virtual machine, as the following virtual machines may inherit the knowledge of the preceding one, which brings additional uncertainty.

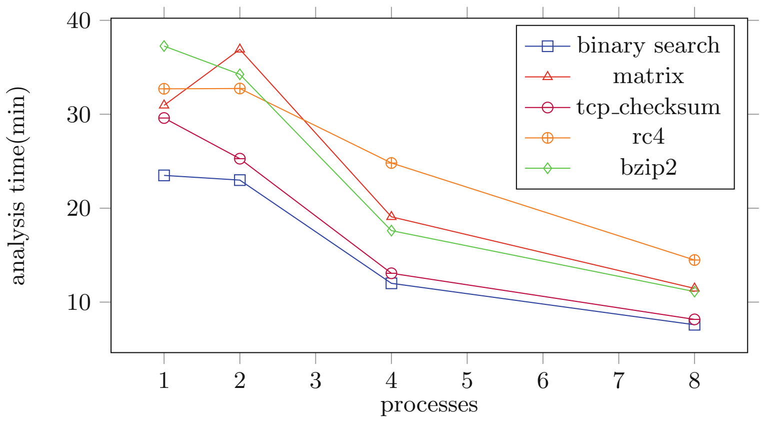


Fig. 3. overhead with multiple process

The Fig. 3 indicates that parallelism does reduce the whole overhead of the analysis. The consuming time nearly comes to 35.52% of the original when using

8 processes at the same time. The whole consuming time decreases with more analysis processes, but not by multiplicative inverse. We found that when using only two processes, the consuming time may be more than using single process. The following reasons may help to understand this phenomenon. First, parallelism brings additional cost with initialization and process communication. Second, some handlers were executed more than once for reducing the cost of communication.

We also compared the efficiency of SymSem with another deobfuscation tools driven by symbolic execution, VMHunt. We collected data from the published paper. We compare the average data based on 6 test cases in VMHunt and 10 in SymSem. We mainly concern two important indexes in the symbolic execution system. From Table 6, We find that though SymSem spends less time on each instruction, its ability to simplify instruction before symbolic execution is not comparable with VMHunt. The most significant reason for this is that VMHunt also puts data dependence analysis into its extraction process before symbolic execution.

Table 6. Efficiency comparison with VMHunt. In this table we compare the code extraction rate and the execution time per instruction of two different tools. The average data is based on 6 test cases in VMHunt and 10 in SymSem. The tag are defined as follows: "SE" = symbolic execution. "CE" = code extracition rates, it indicates how much code would be extracted for symbolic execution compared to the whole trace. "TPI" = symbolic execution time per instruction.

Name	avg_total_trace	avg_SE_len	avg_SE_time	CE	TPI
SymSem	354000.2	41311.0	1112.725 s	11.67%	0.0269 s
VMHunt	2613690.1	2011.3	339 s	0.076953%	0.1685 s

7 Related Works

Deobfuscation of Virtualized Code. The deobfuscation of programs protected by code virtualization has always been a difficult problem. Since Rolles proposed a deobfuscating method based on virtual machine architecture analysis [15], most of the automatic methods aim at virtual machines with dispatcher-handler model [11,12,15,16], one of the which is VMAttack [11]. It applies many heuristic rules to pair handlers with translated mnemonic.

Another type of deobfuscating method aims to sift important instructions from the program trace. The widely used approaches includes dataflow analysis, control flow analysis and taint analysis. These methods have an advantage that they do not rely on any assumptions of virtual machine architectures. One of the representative work is from Coogan [10], which uses equational reasoning to analyze instructions related to system calls. VMHunt [18] and Bin Sim [14] are similar for applying backward slicing based on different sources. Furthermore, VMHunt also use symbolic execution to do semantic analysis on sliced instructions.

Symbolic Execution of Code. Symbolic execution has been one of the fundamental technologies in automatic analysis [5,19]. Assisted by modern powerful SMT solver, symbolic execution abstract the target problem as constraint solving, made great help in automatic exploit generation [6,9], control flow analysis [7]. Also, it is also widely used in deobfuscation. [12,18]. Automatic reverse engineer tools take advantage of symbolic execution to deal with branch conditions, represent semantic results or further generate optimized code for analysis.

However, as we have pointed out, symbolic execution also suffers from alias and other problems, which may cause semantic confusion. We first point out these problems and provide symbolic execution with time stamps to solve them.

Rewrite of Obfuscated Code. Binary rewriting is an kernel techniques in security application area. It is widely used in profiling, code optimization, vulnerability detection. Our work uses a dynamic rewriting approach [13] to evaluate the correctness of trace rewriting. The same methods are also applied in evaluation of many other binary rewriting tools [8,17].

8 Discussion

We now address possible limitations of our method here. First, SymSem is based on dynamic analysis. As its analysis output is limited by the input trace, it is unable to recover the whole semantics of the original function. Second, symbolic execution method is unable to deal with those self modify code. These code have continuously varying semantics which is unable to represented by simply a set of symbolic expressions.

9 Conclusion

Symbolic execution has become a popular technique widely applied in fuzzing, vulnerability exploitation and reverse analysis. We present a new kind of symbolic execution technique which can accurately generate expressions that represent the full semantics of the code. In this symbolic execution technique, time stamps play a important role in eliminating possible hazards and alias issues. Also, by realizing a prototype named SymSem, we prove this technique can be further implemented to assist reverse engineer of programs obfuscated by code virtualization. We expect further costume compilation optimization will recover the original semantics of the unobfuscated program.

Acknowledgments. This work was supported by the General Program of National Natural Science Foundation of China (GrantNo. 61872237).

References

1. Code virtualizer. https://www.oreans.com/codevirtualizer.php. Accessed 4 July 2019
2. Manticore. https://github.com/trailofbits/manticore. Accessed 4 July 2019
3. Themida. https://www.oreans.com/themida.php. Accessed 4 July 2019
4. Vmprotct. https://vmpsoft.com/. Accessed 4 July 2019
5. Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., Pretschner, A.: Code obfuscation against symbolic execution attacks. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, pp. 189–200. ACM, New York (2016). https://doi.org/10.1145/2991079.2991114
6. Bao, T., Wang, R., Shoshitaishvili, Y., Brumley, D.: Your exploit is mine: automatic shellcode transplant for remote exploits. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 824–839, May 2017. https://doi.org/10.1109/SP.2017.67
7. Bardin, S., David, R., Marion, J.Y.: Backward-bounded DSE: targeting infeasibility questions on obfuscated codes, pp. 633–651, May 2017. https://doi.org/10.1109/SP.2017.36
8. Bauman, E., Lin, Z., Hamlen, K.: Superset disassembly: statically rewriting x86 binaries without heuristics. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, CA, February 2018
9. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: techniques and implications. In: 2008 IEEE Symposium on Security and Privacy (SP 2008), pp. 143–157, May 2008. https://doi.org/10.1109/SP.2008.17
10. Coogan, K., Lu, G., Debray, S.K.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: ACM Conference on Computer & Communications Security (2011)
11. Kalysch, A., Götzfried, J., Müller, T.: VMAttack: deobfuscating virtualization-based packed binaries. In: The 12th International Conference (2017)
12. Liang, M., Li, Z., Zeng, Q., Fang, Z.: Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization. In: Qing, S., Mitchell, C., Chen, L., Liu, D. (eds.) ICICS 2017. LNCS, vol. 10631, pp. 313–324. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89500-0_28
13. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005). https://doi.org/10.1145/1065010.1065034
14. Ming, J., Xu, D., Jiang, Y., Wu, D.: BinSim: trace-based semantic binary diffing via system call sliced segment equivalence checking. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 253–270. USENIX Association, Vancouver (2017). https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming
15. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies, January 2009
16. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: IEEE Symposium on Security & Privacy (2009)
17. Wang, R., et al.: Ramblr: making reassembly great again. In: Proceedings of the Network and Distributed System Security Symposium (2017)
18. Xu, D., Ming, J., Fu, Y., Wu, D.: VMHunt: a verifiable approach to partially-virtualized binary code simplification. In: Proceedings of the 2018 ACM SIGSAC

Conference on Computer and Communications Security, CCS 2018, pp. 442–458. ACM, New York (2018). https://doi.org/10.1145/3243734.3243827

19. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 732–744. ACM, New York (2015). https://doi.org/10.1145/2810103.2813663