Deobfuscation of Virtualization-Obfuscated Software

A Semantics-Based Approach

Kevin Coogan
Department of Computer Science
University of Arizona
P.O. Box 210077
Tucson, AZ 85721-0077
kpcoogan@cs.arizona.edu

Gen Lu
Department of Computer Science
University of Arizona
P.O. Box 210077
Tucson, AZ 85721-0077
genlu@cs.arizona.edu

Saumya Debray
Department of Computer Science
University of Arizona
P.O. Box 210077
Tucson, AZ 85721-0077
debray@cs.arizona.edu

ABSTRACT

When new malware are discovered, it is important for researchers to analyze and understand them as quickly as possible. This task has been made more difficult in recent years as researchers have seen an increasing use of virtualization-obfuscated malware code. These programs are difficult to comprehend and reverse engineer, since they are resistant to both static and dynamic analysis techniques. Current approaches to dealing with such code first reverse-engineer the byte code interpreter, then use this to work out the logic of the byte code program. This outside-in approach produces good results when the structure of the interpreter is known, but cannot be applied to all cases. This paper proposes a different approach to the problem that focuses on identifying instructions that affect the observable behavior of the obfuscated code. This inside-out approach requires fewer assumptions, and aims to complement existing techniques by broadening the domain of obfuscated programs eligible for automated analysis. Results from a prototype tool on real-world malicious code are encouraging.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software*

General Terms

Security

Keywords

virtualization, deobfuscation, dynamic analysis

1. INTRODUCTION

Recent years have seen an increase in malware protected against analysis and reverse engineering using virtualization obfuscators such as VMProtect [16] and Code Virtualizer [11]. Such obfuscators embed the original program's logic within the byte code for a (custom) virtual machine (VM) interpreter. It is difficult to recover the logic of the original program because an examination of the executed code reveals only the structure and logic of the byte-code interpreter. Additionally, there may be an element of randomness introduced into the construction of the custom VM, so that successful reverse engineering of one instance of a virtualization-obfuscated program does not help us deal with a different program obfuscated using the same obfuscator. This makes the task of reverse engineering virtualization-obfuscated malware code a challenging one.

Existing techniques for reverse engineering of code protected by virtualization-obfuscation [6, 12, 13] first reverse engineer the VM interpreter; use this information to work out individual byte code instructions; and finally, recover the logic embedded in the byte code program. This outside-in approach is very effective when the structure of the interpreter meets certain requirements. However, when the interpreter uses techniques that do not fit these assumptions (e.g., direct-threading vs. byte-code interpretation), the deobfuscator may not work well. This approach may also not generalize easily to code that uses multiple layers of interpretation, since it may be difficult to distinguish between instruction fetches for various interpreters.

This paper presents a prototype tool that uses a different approach to dealing with virtualization-obfuscated programs. We note that for modern operating systems, programs interact with the system through a predefined interface, typically implemented as system calls. We also note that malicious code must use this interface if its behavior is to be meaningful or impactful in any way. We identify instructions that interact with the system, then use various analyses to determine which instructions affect this interaction, either directly or indirectly. The resulting set of instructions is an approximation of the original code, while the remaining instructions approximate the set of instructions that are semantically uninteresting and can be discarded.

The previous work mentioned above produces excellent results on those programs that match their assumptions. Because our approach does not attempt to recover the original instructions, but rather attempts to capture the relevant behavior of the code, it will not match those results for accuracy. However, our approach is more general, and can be applied to a wider range of obfuscation techniques. Thus, it should be seen as complementing existing approaches by providing information when others cannot.

The remainder of the paper is organized as follows: Section 2 describes the problem in detail and our approach to the analysis, Section 3.1 describes our methodology for evaluation of our results, Section 3.2 presents the results of our analysis, Section 5 discusses related works and Section 6 presents our conclusions.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

2. DEOBFUSCATION

Static analysis of code that has been obfuscated using virtualization reveals only the structure of the virtual machine interpreter. Similarly, the dynamic trace of a virtualization-obfuscated executable is a mix of virtual machine interpreter instructions and instructions performing the work of the original program. It is often difficult to see the boundaries between these two sets of instructions when looking at the trace. This task becomes even harder in the case that multiple interpreters are used, or when the interpreter dispatch routine performs multiple operations (e.g., decrypting the address of the next instruction).

Our approach is to try to identify instructions that are known to be part of the original code, and eliminating those that are not, while not assuming any information about the specific structure of the interpreter or its dispatch routines. In the remainder of this section, we present an outline of our overall approach, then discuss the parts of this approach that are original work in further detail.

2.1 Overall Approach

The analysis of an executable consists of the following steps:

1. Use a tracing tool such as qemu [3], OllyDbg, Ether [5], etc. to obtain a low level execution trace that provides, at each execution step, the address of the instruction executed, details about this instruction (byte sequence, mnemonic, operands, etc.), and the values of the machine registers.
2. Identify system calls and their arguments in this trace, using a database that gives information about arguments and return values of system calls.[1] In general, not all system calls may be of interest (e.g., those occurring in program start up or exit code may not be interesting), so we allow the user to optionally indicate which system calls to consider.
3. Use the available information to carry out analyses on the instruction trace. These analyses flag instructions that affect the values of arguments, as well as conditional control flow and the flow of control to system calls of interest. We refer to these instructions as *relevant instructions*.
4. Build a subtrace from those instructions that have been marked as relevant. This *relevant subtrace* approximates a dynamic trace of the original, unobfuscated code.

2.2 Value-based Dependence Analysis

To motivate our approach to deobfuscation, we begin by considering the semantic intuition behind any deobfuscation process. Obviously, when we simplify an obfuscated program, we cannot hope to recover the code for the original program for two reasons. First, in the case of malware we usually do not have access to the source code. Second, even where source code is available, the program may change during compilation, e.g., via compiler transformations such as in-lining or loop unrolling, so that the code for the final executable may be different from (though equivalent to) that of the original program. All we can require, then, is that the process of deobfuscation must be semantics-preserving: i.e., that the code resulting from deobfuscation be semantically equivalent to the original program.

In the context of malware analysis, a reasonable notion of semantic equivalence seems to be that of *observational equivalence*, where two programs are considered equivalent if they behave—i.e., interact with their execution environment—in the same way. Since a program's runtime interactions with the external environment are carried out through system calls, this means that two programs are observationally equivalent if they execute identical sequences of system calls (together with the argument vectors to these calls).

This notion of program equivalence suggests a simple approach to deobfuscation: identify all instructions that directly or indirectly affect the values of the arguments to system calls; these instructions are "semantically relevant." Any remaining instructions, which are by definition semantically irrelevant, may be discarded. The crucial question then becomes that of identifying instructions that affect the values of system call arguments: we discuss this issue in more detail in the remainder of this section.

The goal of dependence analysis is to work back from system call arguments to identify all instructions that directly or indirectly affect the values of those arguments. At first glance, this seems to be a straightforward application of dynamic program slicing [14], but this turns out to not be the case. The problem is that slicing algorithms follow all control and data dependencies in the code (an instruction I is *control-dependent* on an instruction J if the execution of J can affect whether or not control goes to I). Since the instructions that implement a byte-code operation are all control-dependent on the dispatch code in the interpreter, it follows that the code that evaluates system call arguments and makes the system calls will also be control dependent on the interpreter's dispatch code. The net result is that slicing algorithms end up including most or all of the interpreter code in the computed slice and so achieves little in the way of deobfuscation.

We use a different approach where we initially follow only data dependencies, then consider control transfers separately. We use a variation on the notion of *use-definition* (ud) chains [2]. Conventional ud-chains link instructions that use a variable (register, memory location) to the instruction(s) that define it. While ud-chains are usually considered in the context of static analysis of programs, it is straightforward to adapt them to dynamic execution traces. In this case, we must match each use of a variable with the instance of the instruction in the trace that defines it.

Because they do not follow control dependencies, ud-chains avoid the imprecision problem encountered with program slicing (control flow has to be identified separately in the deobfuscated code.) However, conventional ud-chains have precision problems of their own. Consider the following instruction sequence:

```
/*I1*/  mov eax, [ecx+edx]
/*I2*/  push eax
/*I3*/  call print
```

The argument to the print call is loaded from memory by instruction I_1, then pushed onto the stack by instruction I_2. A conventional ud-chain calculation would show that instruction I_1 uses ecx, edx, and the memory address pointed to by adding these values together. However, only the memory address is relevant to the *value* passed to the system call. This results in a loss of precision. What we should do, instead, is disregard the registers used for the address computation and trace back to find the (most recent) instruction that wrote to the memory location being accessed.

To deal with this issue, we define a notion of *value-based dependence*. The essential intuition here is that we focus on the flow of *values* rather than on details of the intermediate computations of

[1] Our current implementation uses DLL calls as a proxy for system calls, primarily because the Microsoft Windows API for DLLs is better documented and also more consistent across different versions of the Windows operating system. This generally causes the analysis to be sound but possibly conservative since not all DLL calls lead to system calls. It is straightforward to modify this to handle code that traps directly into the kernel without going through a DLL: it simply requires examining the argument values of instructions that trap into the operating system kernel, e.g., sysenter, to determine the syscall number and hence the system call itself.

the addresses of these values. This is done by redefining the set of locations used by an operand as follows:

use(*op*) =
if *op* is a register r **then** $\{r\}$
else if *op* specifies a memory address a **then** $\{a\}$
else $\emptyset$;

We then identify the instructions that are *relevant* to the system calls executed by the program as follows. For each system call in the execution trace, we use ABI information to identify the arguments that are being passed; we initialize a set S to the locations holding these arguments. We then scan back in the trace, starting at the the system call, and process each instruction I as follows: if I defines a location $\ell \in$ S (which may be a register or a memory location) then I is marked as *relevant*, ℓ is removed from S, and the set of locations used by I according to our notion of value-based dependencies (see use() above) is added to S. This backward scan continues until S becomes empty or we reach the beginning of the trace. The effect of the value-based dependence analysis described above is that when an instruction I accesses a value from a memory location a, the dependence analysis works back to find the nearest previous instruction that wrote to location a but ignores the details of how the address a was computed by I.

Under certain conditions, the above algorithm may suffer from a lack of precision. The problem arises when the parameter in question is a pointer to a structure of some sort, and the function call is using an element of that structure. The trace-back based on the pointer itself only reveals the initialization of the structure. Without knowing the size of the structure, we will not recognize when elements of the structure are being set. To solve this problem, prior to performing our analysis, we must analyze the trace of the system calls to identify what values are used, and if those values are referenced using the pointer parameter.

For each system call, we create a set P, which holds all of the locations (register or memory locations) which might potentially be pointers, and a set M, which holds all of the memory locations that have been accessed through a pointer. Initially, P holds the stack locations of the parameters to the call, and M is empty since we have not encountered any uses yet. We scan forward through the trace of the system call and look at each instruction I. Typically, I will uses some number of locations (i.e., register and memory locations), which we will call ℓ_1 , ℓ_2 , . . . , to define some location, which we will call ℓ_d. If I uses some location $\ell_i \in$ P to define ℓ_d, then ℓ_d may potentially also be a pointer and is added to P. Furthermore, if ℓ_i is known to access a memory location (e.g., eax in the instruction "move ebx, [eax]), then the value v stored at ℓ_i is also added to set M, since we know that it is a memory location accessed through a suspected pointer. Finally, if instruction I defines a location $\ell_d \in$ P, and I does not use any values from P, then we can assume that I is redefining ℓ_d as something other than a pointer that we are tracking, and we remove ℓ_d from P. The algorithm continues until P is empty, or until the end of the system call trace is reached. At this point, the set M contains a set of memory locations that we suspect are part of structures pointed to by one of the system call parameters. These locations are added to the set S above as part of the parameters passed to the call.

2.3 Relevant Conditional Control Flow

Value-based dependence analysis identifies the instructions that compute the values of system call arguments, but not the associated control flow instructions. The problem with identifying relevant control transfer instructions in virtualized code is that control transfers may be handled by the same dispatch code that handles other VM instructions. In the popular IA-32 (x86) architecture (the target of our analysis tool), conditional statements are typically implemented by setting the appropriate condition code flags in the designated eflags register, then executing a conditional branch statement, e.g., jnz that reads this register. The target of the branch statement is either the address given in the instruction or the address of the next instruction in the code, depending on the value stored in this eflags register. Hence, to recognize when conditional control flow is occurring, we can examine all control flow statements (e.g., jumps, conditional jumps, calls) to see how their target addresses are calculated. Any control flow instruction whose target address calculation is conditionally dependent on some previous value is an implementation of a conditional branch statement.

It is possible that conditional logic will not be implemented exactly as described above in virtualization-obfuscated code. For example, VMProtect eliminates the branch statements and moves the value of the flags register to other general purpose registers for manipulation. However, while theoretically possible, we are not aware of any obfuscation programs that implement conditional logic without the use of the value of the eflags register at some point in the code. Hence, we can examine target address calculations for any dependence on the value of the eflags register as an indication of conditional dependence. Even assuming that the flags register is used, there are still multiple ways to use this value to implement conditional logic. Thus, our approach must be general enough to handle any such implementation. We handle this problem through the use of an equational reasoning system that was developed in-house to handle x86 assembly code [4].

Our equational reasoning system translates each instruction in the dynamic trace into an equivalent set of equations. We note that in the dynamic trace, there may be multiple equations that define the same register or memory location. To maintain the original behavior of the trace, we number the variables as follows. A variable appearing on the left hand side of an equation (i.e., a variable that is being defined) is numbered according to the order that its instruction appears in the trace. A variable appearing on the right hand side of an equation (i.e., a variable that is being used) is numbered according to the instruction that defined it. These defining instructions are found by searching backwards through the trace to identify where the definition came from.

```
...
/*I10*/   mov ebx, 0x0
/*I11*/   pop eax
/*I12*/   add ebx, eax
/*I13*/   pop eax
/*I14*/   sub ebx, eax
```

(a)

```
...
ebx10 = 0x0
eax11 = ValueAt(M1000)
esp11 = esp8 + 4
ebx12 = ebx10 + eax11
eflags12 = Flag(ebx10 + eax11)
eax13 = ValueAt(M1004)
esp13 = esp11 + 4
ebx14 = ebx12 - eax13
eflags14 = Flag(ebx12 - eax13)
```

(b)

Figure 1: Simple example of translating instructions into equivalent equations.

Consider the example in Figure 1. Figure 1(a) gives a snippet of x86 assembly code, and Figure 1(b) gives the equivalent equations generated by our system. We are assuming that the value of

the stack pointer esp before instruction I_{10} executes is 1000. We use the notation "ValueAt(M1000)" to indicate the value stored at memory location 1000. Notice that the eax used in the equations for instruction I_{12} is not the same as the eax used in the equations for instruction I_{14}, as is indicated by our numbering scheme.

Also notice that we are explicitly handling the setting of the eflags register by generating its own equation. We introduce the "Flag" operation to indicate the calculation of the flag register value based on the expression passed as a parameter. So, for instruction I_{12}, ebx gets the result of adding ebx_{10} to eax_{11}, and the eflags register gets changed according the result of that same operation.

For our purposes here, we are primarily concerned with the calculation of the target addresses of control flow instructions. Specifically, as described, we need to determine if any component of the calculation of such a target address is dependent on the value of some flag calculation. With our equational reasoning system, we need to generate a simplified expression for the target address at the point it is used, then check that expression to see if it contains any calls to the "Flag" operation.

We must also account for the possibility of additional or trivial conditional logic added for the purpose of obfuscation. The obfuscation routine cannot change the behavior of the original program, but it can add branch statements that are always true or always false, to try to confuse analysis. For this reason, we can eliminate any conditional logic that reduces to a constant boolean value.

Next, we will examine several examples of increasing complexity to show how our system correctly identifies these conditional dependencies. First, we look at the simple example in Figure 2(a), where the normal branch instructions are used to implement conditional control flow. We know that the jnz instruction uses the value of the eflags register to decide whether or not to branch, so we add a new equation at the point of the jnz instruction to represent the value of the eflags register.

```
...
/*I10*/   cmp ebx, eax
/*I11*/   mov ebx, 0x0
/*I12*/   mov eax, 0x10
/*I13*/   jnz 10000
```

(a)

```
...
eflags10 = Flag(ebx7 cmp eax6)
ebx11 = 0x0
eax12 = 0x10
eflags13 = eflags10
```

(b)

Figure 2: Identifying control dependencies with branch instructions.

As seen in Figure 2(b), when we trace back to find the definition of the right hand side, we see that it is the value of eflags from instruction I_{10} that is being used. By substituting the definition of $eflags_{10}$, we see that the value of the flags register used by the conditional jump instrution is "Flag(ebx_7 cmp eax_6)."

Next, we consider a case where the standard branch instructions are not used. This case would be anticipated when analyzing virtualization-obfuscated code, since the dispatch routine typically handles all control flow. Consider the code snippet in Figure 3(a). We see that the indirect jump of instruction I_{16} is indexing into a table located at address 0x10000. The value of eax indexes into the table some number of 4-byte values. Instruction I_{10} sets eax equal to some index value to be used. Instructions I_{11} through I_{15} perform some comparison that sets the value of the flags register, then moves the flag value into the ebx register, and masks the value. The effect is as follows. If the result of the comparison turned on the "zero" flag, then the value of the ebx register after instruction I_{14} is one, otherwise, it is zero. The value in ebx is then added to the index value stored in eax such that the actual index value used in the jump depends on the result of the comparison in instruction I_{11}. In the context of virtualization-obfuscated code, the index value is the byte code of the next instruction, and the table contains the addresses of virtual instruction implementations.

```
...
/*I10*/   mov eax, index
/*I11*/   cmp ebx, ecx
/*I12*/   pushf
/*I13*/   pop ebx
/*I14*/   and ebx, 0x1
/*I15*/   add eax, ebx
/*I16*/   jump [eax*4 + 0x10000]
```

(a)

```
...
eax10 = index
eflags11 = Flag(ebx4 cmp ecx3)
esp12 = esp9 - 4
ValueAt(M1000)12 = eflags11
ebx13 = ValueAt(M1000)12
ebx14 = ebx13 & 0x1
eax15 = eax10 + ebx14
target16 = eax15 * 4 + 0x10000
```

(b)

Figure 3: Identifying control dependencies with no branch instructions.

To handle this case, we recognize the control flow statement is an indirect jump and depends on the target address calculation. We generate an equation for the target address calculation as seen in Figure 3(b), and simplify it as described before. The result is given in Figure 4, and shows that the target address depends on the result of the "Flag" operation. Thus, the indirect jump is acting as a conditional control flow statement.

$$target_{16} = index + (Flag(ebx_4 \text{ cmp } ecx_3) \& 0x1)$$

Figure 4: Result of target address simplification from Figure 3.

Finally, we examine a case inspired by the conditional control flow implementation that is used in VMProtect. We begin with the example in Figure 3, and add the use of indirection. In the code snippet of Figure 5(a), two index values are stored to adjacent locations in memory. The same trick is then used to conditionally load either the address of the first index or the address of the second index into the esi register. Finally, the value stored at the location in esi is loaded into the eax register, and the indirect jump calculates the address in the table to use.

For this example, let's assume that it was *index1* that was used in the calculation of the target address, and that the value of *address* was 5000. When we simplify our equation for $target_{20}$, we will substitute "ValueAt(esi_{18})" for "eax_{19}." By our assumption, we know that esi_{18} holds the value 5000, so we substitute the value at memory location 5000 which is *index1* from instruction I_{10}. Our simplified expression for the target is then "$target_{20}$ = *index1* * 4 + 0x10000." From this result, it appears that this jump is not implementing conditional control flow. However, this is wrong. We know from our analysis of the code that the result of the compare instruction determines whether *index1* or *index2* is used to index into the table. The problem is that the conditional element has been hidden by a layer of indirection. When we calculate an expression for the target address, we are only using direct dependencies.

```
...
/*I10*/   mov [address], index1
/*I11*/   mov [address + 4], index2
/*I12*/   mov esi, address
/*I13*/   cmp ebx, ecx
/*I14*/   pushf
/*I15*/   pop ebx
/*I16*/   and ebx, 0x1
/*I17*/   mul ebx, 0x4
/*I18*/   add esi, ebx
/*I19*/   mov eax, [esi]
/*I20*/   jump [eax*4 + 0x10000]
```

(a)

```
...
ValueAt(address)10 = index1
ValueAt(address + 4)11 = index2
esi12 = address
eflags13 = Flag(ebx4 cmp ecx3)
esp14 = esp9 - 4
ValueAt(M1000)14 = eflags13
ebx15 = ValueAt(M1000)14
esp15 = esp14 + 4
ebx16 = ebx15 & 0x1
ebx17 = ebx16 * 0x4
esi18 = esi12 + ebx17
eax19 = ValueAt(esi18)
target20 = eax19 * 4 + 0x10000
```

(b)

Figure 5: Example of code using indirection to hide control dependencies.

To resolve this issue, we must account for any number of layers of indirection. We could analyze the code, as we did here, and recognizing that it was address 5000 that was used, and not 5004. However, this requires understanding how the code works at each step, and will quickly become more difficult as code complexity increases. Instead, we add new equations to our set that define where the calculation of memory addresses come from. We introduce a new variable for each memory access, named with the prefix "LOC" followed by the actual address that was accessed. Then a new equation is added that sets the value of that new variable according to how the calculation was done. Each variable is labeled with the order number as before to guarantee that they are unique. Figure 6 shows the addition of the memory location equation (marked *) for the memory access of instruction I_{19} in Figure 5. In practice, we would add similar equations for all memory accesses. These are omitted here for the sake of clarity.

```
...
ValueAt(5000)10 = index1
ValueAt(5004)11 = index2
esi12 = 5000
eflags13 = Flag(ebx4 cmp ecx3)
esp14 = esp9 - 4
ValueAt(M1000)14 = eflags13
ebx15 = ValueAt(M1000)14
esp15 = esp14 + 4
ebx16 = ebx15 & 0x1
ebx17 = ebx16 * 0x4
esi18 = esi12 + ebx17
(*) LOC_500019 = esi18
eax19 = ValueAt(LOC_500019)
target20 = eax19 * 4 + 0x10000
```

Figure 6: Equations augmented to handle indirection.

Now, for each memory access that is used to calculate the target address, we can start a new simplified expression for the address calculation. If this calculation shows some dependency on the "Flag" operation, then we know that the target address is indirectly conditionally dependent. Any memory access simplified expression that does not show any conditional dependence can be discarded because it is irrelevant. Returning to our example in Figure 5, we calculate all simplified expressions for memory access and get the results shown in Figure 7. Here we see the conditional dependence that we expect, and can mark the indirect jump of instruction I_{20} as implementing conditional control flow.

```
LOC_500019 = address +((Flag(ebx4 cmp ecx3) & 0x1) *4)
target20 = index1 * 4 + 0x10000
```

Figure 7: Result of target address simplification from Figure 5.

To handle multiple layers of indirection, we must account for all dependencies. We use an algorithm that propagates information forward as it scans over the equations in the trace. At each step, all control flow dependencies, direct and indirect, are collected as described above and associated with that equation. Later, if that equation is used in a target address calculation, the complete list of dependencies is known and can be scanned for conditional dependencies. The algorithm is presented in Figure 8.

Input: *List of Equations* : EqnList
Output: *List of simplified expressions* : SimpEqnList

```
Deps = ∅
SimpEqnList = ∅
for each equation in EqnList:
    if IsConditional(equation)
    then
      Deps = Deps ∪ equation
    endif
    MemAccesses = GetMemoryAccessesForEqn(equation)
    for each memAcc in MemAccesses:
      Deps = Deps ∪ memAcc→deps
    endfor
    ExprList = SimpEqnList ∪ equation
    for each term in equation:
      replacement = FindReplacement(EqnList, term)
      if replacement != ∅
      then
        ReplaceTerm(term, replacement)
        Deps = Deps ∪ replacement→deps
      endif
      equation→deps = Deps
    endfor
endfor
return SimpEqnList
```

Figure 8: Pseudocode of simplified expression and conditional dependency identification

2.4 Relevant Call-Return Control Flow

Functions are important mechanisms for code structuring, and are often the basic building blocks of a program during the design phase. Identifying functions during reverse-engineering and deobfuscation, then, would be a useful step towards program comprehension. In the case of unobfuscated, compiler generated code, this can often be a very straightforward task. For example, x86 compatible compilers such as gcc will typically use a standard preamble for functions that saves the base pointer, then points the base pointer to the top of the stack. Knowing this information, one can search through the code for these instructions and subsequently identify the beginning of many functions in the code. However, this technique is only a convention, and in the case where the code is to be purposefully obfuscated, it need not be followed. This section discusses the behavior of function calls and returns in various forms of

obfuscation, their essential properties, and explains the technique we propose for identifying them.

```
1000      call 5000          1000      push 5000
...                          1004      ret
5000      pop eax            ...
5004      mov eax, [abc0]    5000      mov eax, [abc0]
```

(a) (b)

Figure 9: examples of indirect jump

2.4.1 Behavior of Function Calls and Returns

In general, compiler generated, unobfuscated code uses the assembly instructions call and ret for function calls and returns, respectively. However, knowing how these instructions work allows one to use them for other purposes. For example, the call instruction pushes the address of the next instruction onto the stack, then jumps to its target address. The code in Figure 9(a) uses the call to jump to code at another location, then discards the return address, effectively implementing an indirect jump. A similar technique is often used in *position independent code* (PIC-code) by using a call instruction with a target address offset of 0. The effect is that the address of the next instruction is pushed onto the stack. As another example, the ret statement pops an address off the stack, then jumps to it. The code in Figure 9(b) pushes a hard-coded value onto the stack then executes a ret. The effect, as before, is equivalent to that of an indirect jump.

```
1000      call 5000 /* → f*/
1004  /*L*/mov ebx, [eax]
1008      inc ebx
...
5000  /*f*/ mov eax, [abc0]
5004      ret  /* → L*/
```

(a) no obfuscation

```
1000      push 1008
1004      jmp 5000  /* → f*/
1008  /*L*/mov ebx, [eax]
100C      inc ebx
…
5000  /*f*/ mov eax, [abc0]
5004      ret  /* → L*/
```

(b) simple obfuscation

```
1000      push 1008
1004      mov eax, 5000
1008      jmp [eax] /* → f*/
100c  /*L*/mov ebx, [eax]
1010      inc ebx
…
5000  /*f*/ mov eax, [abc0]
5004      pop ebx
5008      jmp [ebx]  /* → L*/
```

(c) another simple obfuscation

```
1000      push 6000
1004  /*f*/ mov esx, [abc0]
1008      ret  /* → L*/
…
6000  /*L*/mov ebx, [eax]
6004      inc ebx
```

(d) extensive obfuscation

Figure 10: Examples of function call/return obfuscation

Similarly, we can use other instructions to simulate the behavior of a function call and return. Figure 10(a) presents an unobfuscated function call/return pair, as might typically be seen in compiler generated code. During execution, the call instruction at address 0x1000 pushes the address of the next instruction 0x1004 onto the runtime stack, then branches to the callee by copying the function address 0x5000 to the instruction pointer. The callee, when finished, pops the return address from the stack into the instruction pointer, resuming execution at the instruction immediately following the initial call. Figure 10(b) shows a semantically equivalent code snippet that uses simple obfuscation. The call instruction has been replaced by a push instruction that saves the return address to the stack, and a jmp instruction that copies the call target to the instruction pointer. Figure 10(c) gives a slightly more obfuscated version of the same code. Here, the target of the call and the return are both saved to a register first, then a jmp instruction moves the value in the machine register to the instruction pointer. These three examples suggest a possible necessary condition for call statements – that is, that they push the address of the textually following instruction onto the runtime stack, then copy the target of the call to the instruction pointer. Existing work [8] uses this assumption to identify function boundaries, and is capable of detecting the call/return pairs in Figures 10(a) - (c).

However, Figure 10(d) gives an example of a call/return pair that does not follow the above assumption. Here, the memory layout has been changed such that the call is executed by control "falling through" to the callee without explicit branching. The return address, which is initially pushed onto the stack, points to an instruction that is no longer adjacent in memory to the call instruction. This code would not be recognized as a call/return pair by existing techniques; however, from a dynamic point of view, it arguably has the identical behavior of the code in Figure 10(b). Furthermore, there is no technical reason that the return address has to be stored on the runtime stack. A program could easily maintain its own stack-like data structure apart from the system stack. In this case, read and write operations on the memory of this separate structure would substitute for push and pop operations in the traditional implementation. Finally, the value saved to memory does not even need to be the actual return address. Rather, it could be derived from the return address by some invertible transformation, and restored to original form at the last minute before the return. We have witnessed exactly these obfuscations in our work analyzing programs obfuscated by VMProtect and CodeVirtualizer.

2.4.2 Identification Approach

In order to correctly identify function call/return pairs in obfuscated code, we must first identify the essential properties of such pairs that do not rely on unnecessary conditions. Traditionally, a call is indicated by pushing the return address onto the runtime stack, and branching to the callee. The return is effected by retrieving this saved address and copying it to the instruction pointer. Our examples in Figure 10 demonstrate that many of these operations, such as saving the return address to the runtime stack, are not necessary but rather convenient conventions used by compilers that do not try to hide the functionality of code. When we try to identify what is common to all cases, it is much more general. We observe that a return address, in some form, must be saved by the caller before execution hits the target function. Furthermore, the return address must be retrieved by the callee, and used so that flow of control begins at this address after execution of the function. This suggests the following semantics:

Call: a code address is saved at the call site.
Return: the saved address is used for a control transfer at the return point.

Notice that calls and returns are defined as a pair of instructions, such that they cannot be identified individually. Based on this definition, the only tie between the call and return is the function's return address. We use this as a necessary condition for function call/return pairs. This approach handles all of the cases presented in Figure 10. However, we point out that our definition does have one, known shortcoming. In the case where a function pointer (i.e., address) is stored in memory, then used later to jump to that function, these instructions will be identified as a call/return pair. Our current results are virtually unaffected by this case, since the original source makes little or no use of function pointers, and the code is compiled on a commercial compiler. However, we are currently working to find a acceptable solution to the problem.

Next, we show why the above condition, while necessary, is not sufficient. Virtualized code presents an additional and significant challenge. First, because the interpreter makes use of the aforementioned techniques such as a simulated stack in place of the runtime stack. More importantly, because the same instruction used by the interpreter to implement byte-code dispatching can be used to implement function calls and returns. Figure 11 presents a fragment of a byte-code dispatching routine generated by VMProtect (instructions unrelated to dispatching logic are ignored). In this example, esi is used as the VM's instruction pointer by the interpreter. Instructions I_1 through I_4 load the encrypted byte-code from esi, decrypt it, then I_5 uses it as an index to locate the encrypted address of a byte-code handling subroutine in a dispatching table. Instruction I_6 decrypts the address, and eventually saves it at the top of the runtime stack. Finally, a ret instruction is used–similar to the example in Figure 10(b)–to jump to the byte-code handling subroutine.

```
/*I1*/   mov al, [esi]
/*I2*/   ror al, 0x4
/*I3*/   add al, 0x3e
/*I4*/   neg al
/*I5*/   mov edx, [eax*4+0x401e34]
/*I6*/   add edx, 0x5216a67c
/*I7*/   mov [esp+0x28], edx
/*I8*/   pushfd
/*I9*/   push dword [esp+0x30]
/*I10*/  ret 0x34
```

Figure 11: Examples of byte-code dispatching in code obfuscated using VMProtect

Typically, a similar set of dispatch instructions is used for each virtual instruction encountered by the interpreter. This code clearly meets the "save and use" of a target address definition that we present above. In this case, each iteration of the interpreter would be identified as a call/return pair, which is clearly not what we want. To eliminate these false positives, we add one additional step based on the concept of relevant instructions introduced earlier. The idea behind identifying instructions that contribute to the value of system calls is to separate the instructions of the virtual machine from the those of the original code. Thus, any call/return pair for which there are no relevant instructions in the call are semantically irrelevant, and can be ignored. We present the previous definition of calls and returns, along with the condition that there be at least one relevant instruction in the function call, as necessary and sufficient conditions for identifying call/return pairs.

While these conditions may be debatable, we argue that functions are not strictly necessary for the implementation of algorithms as computer code. Rather, they are abstractions that allow human programmers to more easily design and implement solutions. As such, it may be possible to generate assembly level code that meets our definition of a call/return pair, that was not intended by the programmer as an actual call to a function. Such cases are inevitable when analyzing the implementation of something that is, by definition, an abstraction. In these cases, we believe that any pairs that meet our definition are as good as intended call/return pairs.

With these conditions, identifying function call/return pairs becomes straightforward: identify all the address save/use pairs as candidates of function call/return pairs, then remove candidates that enclose no relevant instructions (as previously identified by value-based dependence analysis). The algorithm is shown in Figure 12.

2.5 Relevant Dynamic Trace

The final step is building the relevant subtrace. We use order numbers to combine the results of the previous steps in a meaningful way. The order number is a unique number for each instance of an instruction in the original dynamic trace that represents the order that instruction appears. Instructions labeled relevant because they contributed to the value of the system call parameters are added to the relevant subtrace in order. For call/return pairs, we add standard call and ret instructions at the appropriate locations, regardless of the obfuscated implementation. This works well because the original program is typically generated by a compiler using standard conventions. Similarly, for conditional control flow, we add a generic branch statement that will match with any standard branch statements.

```
Input: T : Trace
Output: P : List of identified call/return pairs
for u = sizeof(T) to 1 do:
    if instruction T[u] is indirect jump to address d
      AND T[u] is not marked as SAVE
      AND T[u] is not used as a DLL call
    then
      s = u;
      relevant_count = 0
      while s ≥ 1 do:
            if T[s] is relevant
            then
              relevant_count++;
            if T[s] initially saves d
            then
              mark T[s] as SAVE;
              mark T[u] as USE;
              if relevant_count > 0
              then
                /* call/return found */
                save (s, u) in P;
              break;
            else
              s--;
    else
      continue;
return P;
```

Figure 12: Pseudocode of function call/return identification algorithm

3. EXPERIMENTAL EVALUATION

3.1 Experimental Methodology

The evaluation of our approach to deobfuscation presents several significant problems that must be addressed. In essence, these problems point back to our previous discussion of program equivalence (see Section 2.2). We have argued that *observational equivalence* is a reasonable goal, but testing for such an equivalence can be difficult. It is necessary to identify the system calls, *and* the instructions that affect their parameters. To see why the system calls alone, or the calls and the values of their parameters are not enough, consider the following example. A program that takes 2 integers and outputs their sum will produce the same output as a program that takes two integers and outputs their product, if the inputs to both programs are 2 and 2. In its simplest form, the only system call required is the print statement.

Even if we take into account the relevant instructions, we need to account for them properly. Previous work by Sharif, *et al* [13] has built control flow graphs for the original program and the deobfuscated program to demonstrate similarity between the two. This approach becomes more difficult as the programs get larger and more complex. Furthermore, the idea is less applicable to our work than theirs. They use knowledge of the interpreter to identify where

original instructions are stored in memory. In those cases where their code is applicable, they are able to recover most or all of the original instructions. Since we identify relevant instructions, control flow graphs of our results will not show the structure resulting from things like dead code, or branches not taken.

To further complicate this idea, there is no guarantee that the obfuscator will use the same instructions from the original program. We have seen how VMProtect and CodeVirtualizer rewrite call's as other semantically equivalent instructions. It is possible that obfuscators may rewrite other instructions. For example, the obfuscator may unroll some loops to hide part of the control flow graph, or it may rewrite a multiply operation as a loop of adds so that new control flow structure is found in the obfuscated code. Quantifying these differences likely will be impractical.

Unfortunately, we do not have a perfect solution to the problem, so we present an imperfect solution that we try to tune to what we know about the current state of virtualization-obfuscated code. Our approach is to treat the traces and relevant subtraces as sequences. We can then use known sequence matching algorithms to compare one trace to another. This approach is robust to the idea that we cannot recover the original code precisely. Matching will give us a score for our deobfuscation, regardless of how good our results are. These scores can be compared on a relative basis. While still imprecise, a score that is significantly higher than another should correspond to better matching.

This approach is also fairly flexible, and allows us to handle several of the issues presented by program equivalence. First of all, we know that the current virtualization programs that we examined rewrite library calls and some conditional branches using semantically equivalent instructions. It is a simple matter to replace library call implementations with a call statement at the appropriate place in the trace. Similarly, conditional branch implementations can be replaced with a generic jcc instruction that will match any conditional branch from the original code. Since the original code is compiled by a commercial compiler and will typically use a these standard instructions, this is a reasonable step, and provides good results. This approach also allows us to handle other instances of semantically equivalent instructions. For example, it is possible that an increment instruction could be rewritten as an add instruction. We can build equivalency classes into our matching algorithm as appropriate, so that an increment matches an instruction that adds one. In doing so, we are moving closer to the idea of comparing the behavior of two traces, and not their actual implementation. This idea is more robust and matches the intent behind program equivalence, since these cases truly are equivalent.

In addition to considering instruction operation equivalences, we must also consider how instruction operands are handled. This issue is especially relevant in the context of virtualization-obfuscated code. Due to the nature of the stack based approach used in the obfuscation programs we examined, it is possible, even likely, that the operands of the instructions will be different than in the original program. For example, in the sample files that we tested, VMProtect uses the esi register as the virtual machine instruction pointer. In CodeVirtualizer, the addresses of virtual instructions are always loaded into the al register. In both cases, the values to be operated on are stored on the virtual stack, and popped into machine registers when needed. There is no technical reason why the virtual machine would try to move these operands into the same registers that were used in the original code. To handle this, we cannot include the operands in the matching algorithm. Instead, we use only the opcode (add, call, etc.) to represent the instruction.

Next, we must consider to what we will match our results. We need to generate a trace of the original program on the same inputs. In order to present an unbiased representation of the original program, we must limit the amount of processing and analysis that is done to this trace. At the same time, we do not want to include instructions that may taint our results. As a result, we eliminate all instructions that result from library calls from both the original trace and the obfuscated trace. There are also a number of instructions that are part of the operating system initialization, and are included in every execution trace. We eliminate these instructions from both the original and obfuscated traces.

The matching algorithm itself is straightforward. Like our analysis, we use the knowledge of system calls as a guide. The traces are broken into segments, where a segment includes all instructions up to and including the next system call, or the end of the trace. In the case where the system calls between traces do not match exactly, we use the subset of calls that form a one-to-one correspondence between the two traces. Segments are then matched, and all segments are aggregated. A matching provides a score representing how many instructions from the original trace appear in either the obfuscated subtrace or our relevant subtrace.

As a final step, we wish to calculate how effective our analysis has been. To do this, we must take into account two competing factors. First, our analysis is trying to identify as many instructions from the original trace as possible. At the same time, we are trying to eliminate as many virtual machine instructions as we can. To this end, we present two numbers for each test. The first, which we call the *relevance score*, is the percentage of the instructions from the original trace that are included in the relevant subtrace. The second, which we call the *obfuscation score*, is the percentage of instructions added by the obfuscator that are correctly excluded from the relevant subtrace. It is easy to optimize either of these values individually, but achieving good (i.e., close to 100%) scores for both is difficult, and will provide a fair evaluation of our work.

Taking into account the above discussion and concerns, we present the following methodology for evaluating our analysis:

1. Original source code of a test program is compiled into an executable.
2. A dynamic trace is generated for the original executable on some input set.
3. An original subtrace is generated by including only instructions from the executable module.
4. The executable file is protected using an available virtualization-obfuscation technique.
5. A dynamic trace is generated for the obfuscated version of the executable.
6. We perform our analysis per Section 2 on the obfuscated subtrace, and generate a relevant subtrace.
7. The obfuscated subtrace is matched to the original subtrace, and scores are produced.
8. The relevant subtrace is matched to the original subtrace, and scores are produced.
9. The relevance score and obfuscation score are calculated.
10. The process is repeated for all combinations of virtualization-obfuscation techniques and input test files.

3.2 Experimental Results

To evaluate our analysis, we tested three toy programs simple enough that results could be checked by hand–an iterative factorial implementation, a matrix multiplaction program with double nested loops, and a recursive fibonacci implementation. We also tested two samples of malicious code–BullMoose and hunatcha–whose C source code was available from the VX Heavens web

Table 1: Results for programs obfuscated with VMProtect

Name	Original trace size	Obfuscated trace size	Relevant trace size	relevant matching	Rel. Score	Obf. Score
factorial	92	15365	222	54	58.7%	99.0%
matrx_mult	651	138798	597	345	53.0%	99.8%
fibonacci	151	16438	167	63	41.7%	99.4%
BullMoose	94	6900	376	36	38.3%	95.0%
hunatcha	2226	3327	1347	1347	60.5%	100.0%
md5	2257	77219	5347	1700	75.3%	95.1%

Table 2: Results for programs obfuscated with CodeVirtualizer

Name	Original trace size	Obfuscated trace size	Relevant trace size	relevant matching	Rel Score	Obf Score
factorial	92	172249	48720	56	60.9%	71.7%
matrx_mult	651	1571686	270143	454	69.7%	82.8%
fibonacci	151	223053	18560	102	67.5%	91.7%
BullMoose	94	120982	32817	67	71.3%	72.9%
hunatcha	2226	3881611	1066993	1524	68.5%	72.5%
md5	2257	5732714	2613431	2099	93.0%	54.4%

site [1]. Finally, we tested a simple benchmark utility that performs the md5 checksum. The results of our analysis on VMProtect obfuscated code are shown in Table 1 and the results of our analysis on CodeVirtualizer obfuscated code are shown in Table 2.

Table 1 shows that for most of our test programs, our analysis is able to identify better than half of the original programs instructions, while in all cases eliminating over 90% of the obfuscation introduced by VMProtect. Similarly, Table 2 shows even better results, identifying on average about 70% of original program instructions from CodeVirtualizer obfuscated code. However, our analysis fairs worse, eliminating about 75% of obfuscation instructions on average. Overall, these results are encouraging, since our approach only identifies those instructions that affect the behavior of the program. We anticipate that many of the "missed" instructions are performing functions like allocating memory or initializing data structures, and will not be caught by our analysis under any conditions. Hand analysis reveals that some of the missed instructions result from instruction implementations whose equivalences (see Section 3.1) were not anticipated. For example, some push and pop instructions were replaced with mov.

We note that, unlike the rest of our test programs, we were not able to achieve our results for the md5 benchmark program through totally automated means. The analysis of this program, even on the unprotected version, used excessive amounts of memory and crashed. The problem resulted from some of the return values of early calls to fopen and fread functions being used in later calculations. The resulting expressions were not able to be simplified easily by our equational reasoning system, which resulted in larger and larger expressions being generated late in the dynamic trace. We would also like to note, however, that our equational reasoning system allowed us to identify this problem easily, and also to select these function call return values for manual simplification. We were able to make these simplifications in about an hour, and feel that the results were still worth reporting.

We examined the results by hand, and found the reason for the lower obfuscation scores on CodeVirtualizer files as compared to VMProtect files. CodeVirtualizer uses an interesting technique that artificially creates a dependency between some original program instructions and the virtual machine interpreter instructions. We believe that this may result from the use of constant values that are stored in memory, and manipulated to add obfuscation. For example, instructions that require the constant 1, may instead use a reference to a variable that holds the value 1. Furthermore, if this variable is modified every time it is used (e.g., it is incremented then decremented just before use), this will create an artificial dependency between the use of the value, and all previous instances of manipulation. Of course, the obfuscator cannot change the function of the original program, so we believe these dependencies can be identified. We have had some success using code simplification techniques such as constant propagation and arithmetic simplification, but work on this issue continues.

The results in both tables also show the extraordinary increase in the number of executed instructions for both obfuscators. Our toy fibonacci program, for example, executes 151 instructions in the original trace. However, the VMProtect obfuscated version executes 16,438 instructions, and the CodeVirtualizer obfuscated version executes 223,053.

4. DEFEATING OUR ANALYSIS

The previous discussion begins to answer an obvious question: how would an attacker defeat our analysis? We must assume that once our analysis is known, malware authors will attempt to exploit its vulnerabilities. Our analysis is highly dependent on the output of our equational reasoning system, which attempts to build simplified equations for each instruction in the dynamic trace. In the case of the md5 analysis, we have seen how calls to fopen and fread could not be simplified away, leading to longer and longer expressions later in the trace of both the unprotected and protected versions of the test file. We have similar results when analyzing the CodeVirtualizer protected versions of code, which stores important values in encrypted form in memory, then decrypts these values before use. The decryption routine is difficult to simplify, and if properly chosen, could similarly cause runtime and memory usage issues in the equational reasoning system. Hence, any such algorithm, embedded in the code, that cannot be simplified away has the potential to cripple our analysis by making the problem impractical.

We have also seen how CodeVirtualizer builds artificial dependencies between the virtual machine code and the original program

code. Since our analysis is based on the idea of separating these two sets of instructions, successfully building such dependencies has the effect of thwarting the analysis. If done properly, it may be possible to build such dependencies between most, or all, instructions in the trace, thus driving our obfuscation scores toward 0%. Because the obfuscation cannot change the function of the original program, we speculate that such dependencies can, in theory, be identified and handled. However, to date, we have not explored this idea in depth, and leave such analysis for future work.

5. RELATED WORK

The deobfuscation of code obfuscated using virtualization obfuscators has been discussed by Rolles [12], Sharif *et al.* [13], and Falliere [6]. These works follow the outside-in approach outlined in Section 1. Lau discusses the use of dynamic binary translation to deal with virtualization obfuscators [9].

There has been some work, in the programming language community, on using a technique called *partial evaluation* [7] for code specialization, in particular for specializing away interpretive code. However, the literature assumes that the program analysis and transformation are static, which suggests that its application to highly obfuscated malware binaries may not be straightforward.

The notion of obfuscation through virtualization has some similarities with the idea of *control flow flattening* [17]. Udupa *et al.* discuss techniques for deobfuscating code that has been subjected to this transformation [15]. These techniques are static, and therefore very different from the ideas presented here.

There is a rich body of work on various sorts of dependence analysis in the program analysis literature. The notion of ud-chains for relating uses and definitions of variables during static program analysis is well-established [2]. There is an extensive body of literature on program slicing [14], but as discussed earlier this technique seems too imprecise for our needs.

6. CONCLUSIONS

Virtualization-obfuscated programs are difficult to reverse engineer because examining the machine instructions of the program, either statically or dynamically, does little to shed light on the program's logic, which is hidden in the interpreted byte-code. Prior approaches to reverse-engineering virtualization-obfuscated programs typically work by first reverse engineering the byte-code interpreter, and then working back from this to work out the logic embedded in the byte code. This paper describes a different approach that focuses on identifying the flow of values to system call instructions. This new approach can be applied to a larger number of obfuscated binaries because it makes fewer assumptions about the nature of the virtual machine interpreter, and still produces good results on test files including two malware executables and one benchmark utility. On average, we identify 50% to 75% of instructions from the original program, while eliminating approximately 75% to 90% of instructions added by the obfuscator.

The system works by gradually adding instructions that are calculated to be of importance, and adding these to a relevant subtrace that represents the behavior of the original, unobfuscated program. Instructions that contribute to the values of system call arguments are included, as are call/return pairs, and conditional control flow statements. We note that our analysis only identifies conditional control flow statements of the original program that appear in the dynamic trace, and, thus, misses code paths that are not executed. In principle, it should be possible to extend multi-path analysis techniques (see, e.g., [10]) to the conditionals so identified. In practice, our intuition is that this will likely be challenging because virtualization-based obfuscation will mean that identifying path constraints (as in Moser et al's work) will not be straightforward. The issue is beyond the scope of this work, but would be interesting and relevant for future work.

Acknowledgements

This work was supported in part by the National Science Foundation via grant no. CNS-1016058, as well as by a GAANN fellowship from the Department of Education award no. P200A070545.

7. REFERENCES

[1] VX Heavens, 2011. `http://vx.netlux.org/`.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1985.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[4] K. Coogan and S. Debray. Equational reasoning on x86 assembly code. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 2011.

[5] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 51–62, 2008.

[6] N. Falliere. Inside the jaws of Trojan.Clampi. Technical report, Symantec Corp., Nov. 2009.

[7] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[8] A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968, 2005.

[9] B. Lau. Dealing with virtualization packer. In *Second CARO Workshop on Packers, Decryptors, and Obfuscators*, May 2008.

[10] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[11] Oreans Technologies. Code virtualizer: Total obfuscation against reverse engineering, Dec. 2008. `http://www.oreans.com/codevirtualizer.php`.

[12] R. Rolles. Unpacking virtualization obfuscators. In *Proc. 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*, Aug. 2009.

[13] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proc. 2009 IEEE Symposium on Security and Privacy*, May 2009.

[14] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[15] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proc. 12th IEEE Working Conference on Reverse Engineering*, pages 45–54, Nov. 2005.

[16] VMProtect Software. Vmprotect software protection, 2008. `http://vmpsoft.com/`.

[17] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.