2015 IEEE Symposium on Security and Privacy

A Generic Approach to Automatic Deobfuscation of Executable Code

Babak Yadegari Brian Johannesmeyer Benjamin Whitely Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
{babaky, bjohannesmeyer, whitely, debray}@cs.arizona.edu

***Abstract*—Malicious software are usually obfuscated to avoid detection and resist analysis. When new malware is encountered, such obfuscations have to be penetrated or removed ("deobfuscated") in order to understand the internal logic of the code and devise countermeasures. This paper discusses a generic approach for deobfuscation of obfuscated executable code. Our approach does not make any assumptions about the nature of the obfuscations used, but instead uses semantics-preserving program transformations to simplify away obfuscation code. We have applied a prototype implementation of our ideas to a variety of different kinds of obfuscation, including emulation-based obfuscation, emulation-based obfuscation with runtime code unpacking, and return-oriented programming. Our experimental results are encouraging and suggest that this approach can be effective in extracting the internal logic from code obfuscated using a variety of obfuscation techniques, including tools such as Themida that previous approaches could not handle.**

***Keywords*-Deobfuscation; Virtualization-Obfuscation; Return Oriented Programming**

I. Introduction

Malicious software are usually deployed in heavily obfuscated form, both to avoid detection and also to hinder reverse engineering by security analysts. Much of the research to date on automatic deobfuscation of code has focused on obfuscation-specific approaches. While important and useful, such approaches are of limited utility against obfuscations that are different from the specific ones they target, and therefore against new obfuscations not previously encountered. We aim to address this problem via a generic semantics-based approach to deobfuscation; in particular, this paper focuses on two very different kinds of programming/obfuscation techniques that can be challenging to reverse engineer: *emulation-based obfuscation* and *return-oriented programming*.

In emulation-based obfuscation, the computation being obfuscated is implemented using an emulator for a custom-generated virtual machine together with a byte-code-like representation of the program's logic [1]–[4]. Examination of the obfuscated code reveals only the emulator's logic, not that of the emulated code. Existing techniques for reverse engineering emulation-obfuscated code first reconstruct specifics of the virtual machine emulator, then use this to decipher individual byte code instructions, and finally recover the logic embedded in the byte code program [5]. Such approaches typically make strong assumptions about the structure and properties of the emulator and may not work well if the analyzer's assumptions do not fit the code being analyzed, e.g., if parts of the emulator are unpacked at runtime [4] or if there are multiple layers of interpretation with distinct virtual program counters that are difficult to tease apart. The work of Coogan *et al.* [6] has similar goals to us, but is based on equational reasoning about assembly-level instruction semantics, which is technically very different from our work (see Section VI) and has the shortcoming that controlling the equational reasoning process can be challenging, making it difficult to recover the logic of the underlying computation into a program representation such as control flow graphs.

A second class of programs that can be challenging to reverse-engineer are *return-oriented programs* (ROP) [7], [8]. While originally devised to bypass defenses against code injection, this programming technique can result in highly convoluted control flow between many small gadgets, leading to program logic that can be tricky to decipher. Other than the work of Lu *et al.* [9], there has been little work on automatic deobfuscation of ROPs.

This paper describes a generic approach to deobfuscation of executable code that is conceptually simpler and more general than those described above. Obfuscation-specific approaches have the significant limitation that they can only be effective against previously-seen obfuscations; they are, unfortunately, of limited utility when confronted by new kinds of obfuscations or new combinations of obfuscations that violate their assumptions. Our work on generic deobfuscation is motivated by the need for deobfuscation techniques that can be effective even when applied to previously unseen obfuscations. The underlying intuition is that the semantics of a program can be understood as a mapping, or transformation, from input values to output values. Deobfuscation thus becomes a problem of identifying and simplifying the code that effects this input-to-output transformation. We use taint propagation to track the flow of values from the program's inputs to its outputs, and semantics-preserving code transformations to simplify the logic of the instructions that operate on and transform values through this flow. We make few if any assumptions about the nature of the any obfuscation being used, whether that be emulation, or ROP, or anything else. Experiments using several emulation-obfuscation tools, including Themida, Code Virtualizer, VMProtect, and ExeCryptor, as well as a number of return-oriented implementations of programs, suggest that the approach is helpful in reconstructing the logic of the original program.

DOI 10.1109/SP.2015.47

II. Background

A. Emulation-based Obfuscation

In emulation-based obfuscation, a program P is represented using the instruction set of virtual machine V_P and interpreted using a custom emulator I_P for V_P. A common representation choice for P is as a sequence of byte code instructions B_P for V_P, where the emulator I_P uses the familiar fetch-decode-execute loop of byte-code interpreters; however, other interpreter implementations, such as direct or indirect threading, are also possible. The instruction set for V_P can be perturbed randomly such that different instances of V_P look very different even if the program P does not change. Further, emulation can be combined with other obfuscations, such as run-time code unpacking, to further complicate analysis.

Reverse engineering of emulation-obfuscated code is challenging because examining the code for the emulator I_P reveals very little about the logic of the original program P, which is actually embedded in the byte-code program B_P. For example, an execution trace of the emulator I_P on the byte-code program B_P will show only the instructions in the emulator I_P. Memory accesses in this trace will contain a mixture of the data manipulation behavior of the original program P and memory operations pertaining to the operation of the emulator I_P; teasing these apart to isolate the memory operations of only the original program P, or only the emulator I_P, can be challenging. Control transfers in the trace, similarly, will be a mixture of those stemming from the logic of P and those corresponding to the dispatch loop of I_P.

B. Return-Oriented Programming

Return-oriented programming (ROP) was introduced as a way to bypass Data Execution Prevention and other defenses against code injection attacks [7], [8]. It uses a multitude of "gadgets," which are small snippets of code ending in *return* instructions, that are present in the existing code in a computer system, whether in the kernel, libraries, or running applications. Each gadget achieves a small piece of computational functionality. The gadgets are strung together by writing their addresses as a contiguous sequence into a buffer that is then used to effect a chain of *return* actions: each *return* then causes the invocation of the next gadget in the buffer. This basic idea has been generalized in various ways to obviate the need for explicit *return* instructions [10], [11].

There are a number of characteristics of ROPs that can make reverse engineering challenging. The first is that the code for a ROP can be scattered across many different functions and/or libraries, making it difficult to discern the logical structure of the code. If these libraries employ Address Space Layout Randomization, or are loaded into dynamically allocated memory, they may occur at different addresses. ROP sequences can take advantage of this fact by being generated just-in-time for the attack, making it difficult to examine what the ROP sequence will do without knowledge of the memory space of the target machine. Secondly, since ROP gadgets are constructed opportunistically from whatever code is already available on a system, they may contain "useless" instructions (from the gadget's perspective) that can be tolerated as long as they do not interfere with the desired functionality of the gadget. However, this opens up the possibility that the same gadget can be invoked in different ways at different times, where a given instruction within the gadget may serve a useful purpose in some invocations and be useless in others. Finally, gadgets can overlap in memory in ways not usually encountered in ordinary programs.

C. Threat Model

Our threat model assumes that the adversary knows our semantics-based approach to deobfuscation as described in this paper, as well as some—but not necessarily all—of the transformation rules used for trace simplification. The latter assumption is justified by the fact that our approach is parameterized by the set of transformation rules used, and these rules do not form a static set but can be augmented with new rules as needed or desired.

III. Our Approach

We use the term *deobfuscation* to refer to the process of removing the effects of obfuscation from a program—i.e., given an obfuscated program P, analyzing and transforming the code for P to obtain a program P' that is functionally equivalent to P but is simpler and easier to understand.

A. Overview

Any approach to deobfuscation needs to start out by identifying something in the code (or its computation) as "semantically significant;" this is then used as the basis for subsequent analysis. For example, when disassembling obfuscated binaries, Kruegel *et al.* begin by identifying control transfer instructions [12]. Automatic unpacking tools such as Renovo [13] look for memory locations that are written to and then executed. More directly relevant to this work, Sharif *et al.* use memory access characteristics of emulation-obfuscated code to identify the emulator's virtual program counter, which they then use to reverse-engineer the emulator [5]. Typically, such notions of semantic significance are based on specific aspects of the code that are either preserved by the obfuscation (e.g., control transfers in the work of Kruegel *et al.* [12]) or else are introduced by the obfuscation (e.g., write-then-execute memory locations for unpacked code [13], emulator components in the work of Sharif *et al.* [5]). In each case, the notion of what constitutes semantically significant code, and the process of identifying such code, is intimately tied to the particular obfuscation(s) being considered.

While such obfuscation-specific assumptions may simplify the process of deobfuscation, they have two drawbacks. First, such assumptions limit the future applicability of the deobfuscation technique to new and as-yet-unseen types of obfuscation. Second, they may provide an adversary a point of attack against the deobfuscation technique by perturbing existing obfuscation techniques in a way that violates the

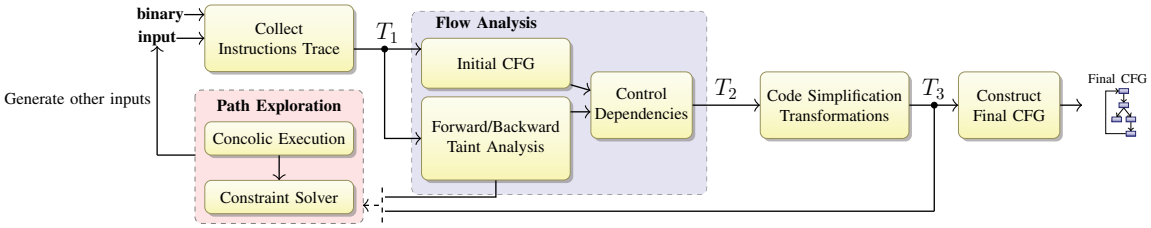


Fig. 1. Overview of the deobfuscation. T_1 is the original trace, consisting of instructions and register values. T_2 is a trace with taint analysis and control dependence information and T_3 is a simplified trace from which a final control flow graph is constructed.

assumptions (an example we encountered recently is illustrated in Figure 10 and discussed in Section VI).

In our case, therefore, we want to minimize assumptions about the obfuscated code; in particular, we do not want to presuppose that any particular kind of obfuscation is being used. Since the identification of semantically significant code is typically closely tied to the obfuscations under consideration, this poses a quandary: what can be considered significant without making assumptions about what obfuscations are being used? To address this, we take an approach inspired by a notion of program semantics where programs are seen as mappings, or transformations, from inputs to outputs [14]. Since malicious code often involves self-modifying and/or dynamically unpacked code, which is difficult to analyze statically, we use dynamic analysis: we collect one or more execution traces of the program, then analyze and simplify these traces. Our approach consists of the following steps:

a) Identifying Input and Output Values: We consider the notion of "input" broadly so as to comprise values obtained from the command line and execution environment of the process (e.g., the Process Environment Block, which is sometimes used by malware to check whether it is being debugged or otherwise monitored, e.g., see [15]) as well as those obtained via explicit input operations; similarly, the notion of "output" is considered to be any externally observable side effect (e.g., creation or deletion of files or processes) as well as the results of explicit output operations and computations.

In our current prototype implementation, input and output values are determined as follows. Any value that is obtained from the command line, or which is defined (written) by a library routine and subsequently read by an instruction in the program, is treated as an input value; any value that is defined (written) by an instruction in the program and subsequently read by a library routine is treated as an output value.[1] Our dynamic analysis environment, which uses a modified version of Ether [16], collects execution traces for library routines as well as the main program, and the flow of values written within the program and subsequently read within a library routine, or vice versa, can be determined by examining the trace. We use a combination of taint propagation and control-dependence analysis to identify instructions in the execution trace that are influenced by input values and/or influence output values.

b) Forward taint propagation: After identifying Input sources, we should propagate the input taint through the trace to find all the instructions which are influenced by input values. In order to do this, we use a taint propagation technique which is a well-known and useful analysis tool in the fields of static and dynamic analysis. It turns out that a conventional byte-level taint analysis is not precise enough for our needs, so we use an enhanced bit-level taint-analysis [17]. This initial computation captures explicit information flow from input to output, but does not capture implicit flows, i.e., associations between data values that arise due to control dependencies rather than data dependencies. To this end, we use dependence analysis to identify control dependencies, which we then combine with the explicit data dependencies identified earlier to capture implicit as well as explicit flow of information from inputs to outputs.

c) Code Simplification: Once we have identified the input-to-output value flows, we iteratively apply semantics-preserving code transformations to simplify the execution trace. The resulting simplified trace represents the behavior of a program that is functionally equivalent to the original program (at least for the particular execution that was observed) but which is simpler.

d) Control Flow Graph Construction: The simplified trace is used to construct a control flow graph (CFG) that makes explicit some of the higher-level control flow structures such as conditionals and loops. The final step of our deobfuscation process is to apply semantics-preserving transformations to the CFG to eliminate some spurious execution paths and produce a more precise CFG. The resulting simplified CFG is then produced as the output of our deobfuscation system.

As mentioned before, dynamic analysis is more powerful when dealing with self-modification or run-time code unpacking but we also need to address the possible low-code coverage issue resulting from recording one execution path. In order to solve this problem we have implemented a concolic execution system which can operate on a trace and produce constraints to solve for other possible inputs to the program to record other execution paths. We can feed both the obfuscated trace and the simplified one to produce alternative inputs to the program. There have been many studies on symbolic execution

[1]This is an over-approximation, since not all library routines interact with the program's execution environment, and so may sometimes lead to a loss in precision of analysis. However, it is conservative.

and multi-path exploration techniques (e.g [18]–[21]) so we do not discuss the details here. Figure 1 gives a high level overview of our approach. We discuss each of these steps in more detail below.

B. Identifying Input-to-Output Flows

The first step of our algorithm is to identify the flow of values from input operations to output operations, and thereby the instructions that transform input values to output values. To this end, we first use taint propagation to identify the explicit flow of values from inputs to outputs, then use control dependence analysis to identify implicit flows.

1) Taint Analysis: Taint analysis finds many important applications in dynamic security analysis. We use it to identify the runtime flow of values from a program's inputs to its outputs; this information is then used for control dependency analysis. The essential idea is to associate each value computed by the program with a bit indicating whether or not it is "tainted," i.e., derived directly or indirectly from an input value. Initially, only values that are obtained directly from inputs are marked as tainted. Taint is then propagated iteratively to other values by marking any value that is computed from a tainted value as tainted. There is a considerable body of literature on taint analysis (e.g., see the paper by Schwartz *et al.* [22]) so we omit the details of the algorithm.

Our approach uses two kinds of taint analysis:

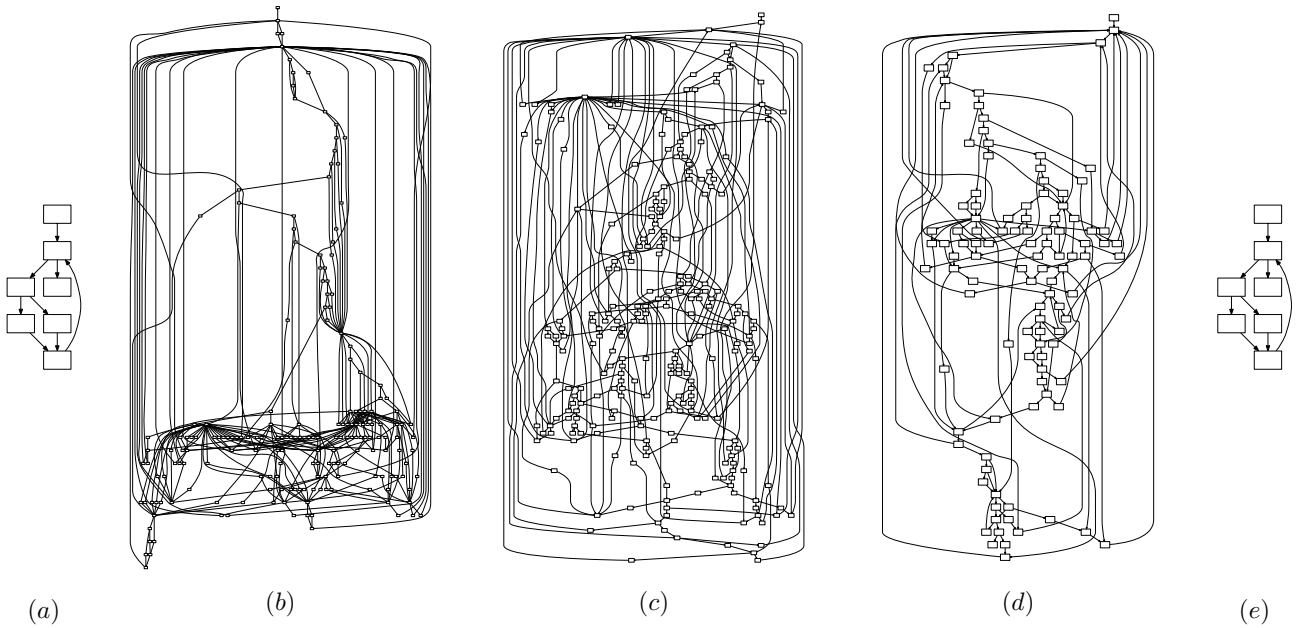
1) *Forward taint analysis*. This is used to identify the flow of input values through the program. It is especially important for finding code that is control dependent on input values. We perform taint analysis for registers, memory, and condition-code flags.
2) *Backward taint analysis*. This starts from output values and works backwards identifying variables and values that influence the program's outputs. In some ways this resembles dynamic program slicing where the slicing criterion is the program's observable output. This is important because static statements under dynamic controls which affect the output should not be simplified away.

The precision of the forward taint analysis is particularly important because the rest of the deobfuscation depends significantly on how well the taint analysis identifies the decision points in the program being examined. As discussed in more detail later, when simplifying the code it is important to identify static computations whose iteration counts are influenced by dynamic input, e.g. loops where the iteration is determined by input values, and imprecision in taint propagation adversely affects the deobfuscation of such loops, e.g., under-tainting leads to too much of the code getting simplified away, and over-tainting leads to too little simplification.

It turns out that traditional byte- or word-level taint analysis is too imprecise for our needs and can result in significant over-tainting. To address this problem, we use an enhanced taint-analysis that differs from conventional taint analyses in two ways. First, in order to deal with obfuscated code—including obfuscations that scramble together the bits from different words—we maintain and propagate taint information at the level of individual bits. Second, instead of simply indicating taintedness via a single bit, indicating whether or not a location is tainted or not, we keep track of the source of each distinct taint value [17]. Keeping track of taint sources turns out to be very helpful for reasoning about the taint of the result of an operation where both inputs originate from the same value; it turns out that such operations are often used in obfuscated code to construct opaque predicates or constants [23]. The propagation of taint values is conceptually analogous to traditional taint analysis, though arithmetic operations have to be handled carefully, e.g., a single tainted bit in a source operand for an **`add`** instruction can cause several bits to become tainted in the result due to carry propagation. This enhanced taint analysis Indeed, ROPs frequently use the carry flag for conditional statements.

As mentioned earlier, the precision of the forward taint analysis algorithm is particularly important for our approach to deobfuscation. Figure 2 illustrates the impact of different taint propagation algorithms on the quality of deobfuscation. The input program is a simple binary search routine whose control flow graph is shown in Figure $2(a)$. The control flow graph of the program resulting from obfuscating this code using a commercial obfuscation tool named ExeCryptor [2] is shown in Figure $2(b)$. Figure $2(c)$ shows the effect of deobfuscation using traditional byte-level taint analysis: this can be seen to be only marginally better that Figure $2(b)$, indicating that the taint propagation is of limited utility. When a bit-level taint analysis is used, the quality of deobfuscation improves considerably, as shown by the control flow graph in Figure $2(d)$; however, although this control flow graph is much simpler than that of Figure $2(c)$, it can be seen to still be significantly more convoluted than the original control flow graph of Figure $2(a)$. However, using our enhanced bit-level taint analysis, which tracks taintedess together with taint source information at the level of individual bits, the deobfuscation process yields much better results, as shown by the control flow graph of Figure $2(e)$.

2) Control Dependency Analysis: Given two instructions (statements) I and J in a program, J is said to be *control-dependent* on I if the outcome of I determines whether or not J is executed. More formally, J is control dependent on I if and only if there is a non-empty path π from I to J such that J post-dominates each instruction in π except I [24]. The identification of control dependencies has been well-studied in the compiler literature [24]. However, the situation is a little different in our case since, because when dealing with emulation-obfuscated code, some of the control transfers encountered correspond to the logic of the program being emulated while others are simply an artifact of the emulation process and therefore not interesting from the perspective of identifying dependencies. We want to find control dependencies of the original program, but we cannot do this simply by examining the control flow graph of the emulator, so we need to untangle the emulator's control flow structure apart from that of the original program.



Key:

(a)	Original program
(b)	Obfuscated program
(c)	Deobfuscation result: traditional byte-level taint analysis
(d)	Deobfuscation result: bit-level analysis (taintedness information only)
(e)	Deobfuscation result: enhanced bit-level analysis (taintedness + taint source information) (our algorithm)

Fig. 2. Impact of different taint analysis algorithms on quality of debofuscation (Input program: binary search; obfuscated using: ExeCryptor)

Algorithm 1: Finding Control Dependencies

Input: An initial input/output tainted trace T
Result: The trace T with control dependencies between instructions identified

1 Construct an initial control flow graph G
2 Compute post-dominator relations in G [24]
3 Use post-dominator relationships to compute explicit control dependencies [24]:
4 (a) C = the set of input-tainted conditional control transfers; and
5 (b) DepVars =$\{x \mid \exists C \in$ C: x control dependent on $C\}$
6 **while** $\exists$ *an indirect control transfer* Ins *dependent on some* $x \in$ DepVars **do**
7 BBl $\leftarrow$ basic block of Ins in G
8 Mark BBl as dependent on the direct control transfer in C that x is dependent on
9 **end**

The approach we take is shown in Algorithm 1. We consider two types of control flows: *explicit* and *implicit*. Explicit control flows are those control transfers where the predicate is explicitly reflected in the transfer of control, e.g., as in conditional jump instructions. Finding explicit control dependencies is straightforward using post-dominators [24]. Implicit control flows are those indirect control transfers of the form '`jmp [`ℓ`]`' where the location ℓ is data-dependent on the set DepVars of dependent variables identified in Algorithm 1. Intuitively, implicit control dependencies account for the fact that a control dependence between two instructions I and J may arise indirectly through an assignment D of the value of a variable x if D is control dependent on I and where x determines the target of an indirect control transfer to J (this happens in, but is not restricted to, the dispatch jump of an emulator).

Figure 3 shows an example of explicit and implicit control flows. The value of register `eax` on line 6 is dependent on the conditional jump on line 2, so the target of the `jmp` instruction of line 6 also depends on which path is taken on line 2. This way the basic block following the `jmp` on line 6 is also control dependent on the conditional transfer on line 2. It is fair to say that the data dependency from line 6 to lines 3 and 5, through the value of `eax`, is really a control dependency in disguise.

C. Trace Simplification

Once we have identified the instructions in the trace that participate in computing output values from input values, the next step is to map these instructions to an equivalent but simpler instruction sequence. Since we want to make as few

```
1        test  ecx, eax
2        jnz   L1
3        mov   eax, 0
4        jmp   L2
5   L1:  mov   eax, 1
6   L2:  jmp   [edx+4*eax]
```

Fig. 3. An example of implicit control flow

assumptions as possible about the obfuscations we may be dealing with, we use a set of simple and general semantics-preserving transformations for this.

An important concept in this context is the notion of a quasi-invariant location. We define a location ℓ to be *quasi-invariant* for an execution if ℓ contains the same value ℓ_c at every use of ℓ in that execution. For constant propagation purposes, we consider a value to be a constant during an execution if either it is an immediate operand of an instruction or if it comes from a memory location that is quasi-invariant for that execution.

Quasi-invariant locations allow us to handle transient modifications to the contents of memory locations, e.g., due to unpacking, as long as we see the same value each time a location is used. Quasi-invariants can be identified in a single forward pass over a trace keeping track of memory locations that are modified and, for each such modification, the value that is written. The notion of quasi-invariance can be extended in various ways, e.g., we may consider whether a memory word contains the same value every time it is used for an indirect branch (this is useful, for example, for dealing with jump tables whose elements are kept in encrypted or encoded form, decrypted prior to use, and then re-encrypted).

The transformations we use include the following (this is a non-exhaustive list):

1) *Arithmetic simplifications*. In essence this is a straightforward adaptation of the classic compiler optimization of constant folding to work with dynamic traces and quasi-invariant locations. However, as described below, it has to be controlled to avoid over-simplification. For example, in the code sequence shown above, the constant value `0xa4` loaded into the register `bh` can be propagated through the bit-manipulation instructions following it, and the entire sequence of instructions manipulating `bh` can be replaced by a single instruction '`mov bh, 0x8b`'.
2) *Indirect memory reference simplification.* An indirect memory reference through a quasi-invariant location ℓ that holds a value A is simplified to refer directly to A. This transformation is applied to both control transfers and data references.
3) *Data movement simplification.* We use pattern-driven rules to identify and simplify data movement. For example, one of our rules states that the following simplification can be performed provided that the sequence of instructions *Instr* does not access the stack and does not change the value of A:

   ```
   push A
   Instrs      ⟶   Instrs
   pop B           mov B, A    /* B := A */
   ```
4) *Dead code elimination.* Instructions whose destinations are dead, i.e., not used subsequently in the computation, are deleted. This transformation must consider all destinations of an instructions, including destination operands that are implicit and which may not be mentioned in the instruction (such implicit destinations includes condition flags).
5) *Control transfer simplification* Control transfer instructions whose targets are constant are replaced by direct jumps. Candidates for this transformation include *return* instructions to constant targets in ROP code as well as indirect jumps to fixed targets in emulation-based obfuscation. Using control flags implicitly to control the transfer flow of the program is common among interpreters and is also used in ROPs. For example one can implement loops in ROPs as follows:

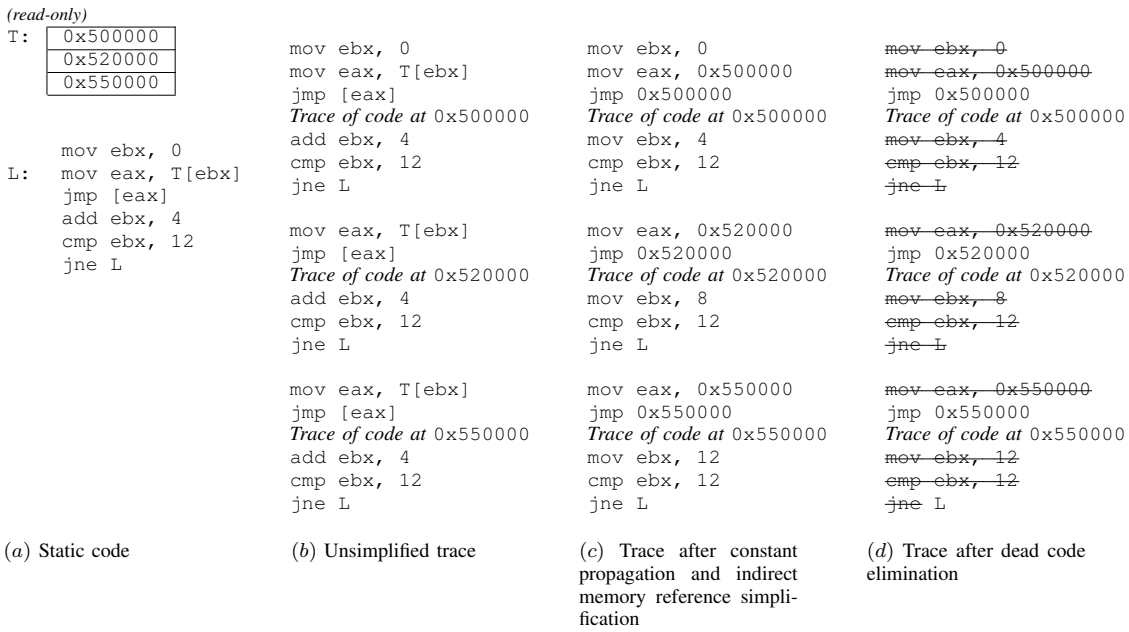
   ```
   mov eax,0
   sub counter,1
   adc eax,eax       /* eax := 1 if counter=0 */
   push [L+eax*4]
   ret
   ```

   where L is the address of the memory location which points to the beginning of the loop and subsequent location points to where loop should exit to. In this example, the target of the return instruction is affected by the outcome of carry flag so the `ret` instruction can be replaced by a conditional jump which directly uses the carry flag.

Example 3.1: Figure 4 gives an example of indirect memory reference simplification. Figure 4(a) shows a small program that sits in a loop making indirect jumps through successive elements of a read-only array `T`*. Figure 4(b) shows the unsimplified trace for this code. Since* `T` *is read-only, its elements are constant, making indirect calls through this table amenable to indirect memory reference simplification; the resulting trace is shown in Figure 4(c). Since* `T` *is no longer being used for indirect jumps, instructions that load from* `T` *then become dead and are removed via dead code elimination. Similarly, constant propagation converts the* `add` *instructions into* `mov` *instructions that load constants into register* `ebx`*. This then determines the outcome of each of the* `cmp` *instructions, and allows the* `cmp` *and* `jne` *instructions to be simplified away; once this happens the instructions that load into* `ebx` *also become dead and are removed.*

The final simplified trace is shown in Figure 4(d). What is left is pretty much just the code executed at the addresses that, in the original program, had been reached via a sequence of indirect jumps through the jump table `T`*. In the simplified trace, almost everything other than the code eventually executed has been simplified away.*

The indirect jump behavior illustrated in this example is very similar to the dispatch code of an emulator. Indirect memory reference simplification allows us to replace the dispatch jumps of an emulator with direct jumps that can



(a) Static code

(b) Unsimplified trace

(c) Trace after constant propagation and indirect memory reference simplification

(d) Trace after dead code elimination

Fig. 4. An example of indirect memory reference simplification

then be candidates for further optimization. Importantly, this is being done via a completely general transformation that makes no assumptions about whether or how an emulator might be dispatching code. ∎

While the trace simplification process described above is crucial for removing obfuscation code, it has to be carefully controlled so that it does not remove too much of the logic of the computation. The problem is illustrated by Figure 5. Figure 5(a) shows the static code for a simple iterative factorial computation, written in a C-like notation for ease of understanding. Figure 5(b) shows the execution trace for this program for an input value of 3, with input-tainted instructions shown underlined. Figure 5(c) shows the result of trace simplification: it can be seen that constant propagation has been applied to all of the updates to the variables `fact` and `i`, and as a result the output operation at the end has been reduced to '`write(6)`'. This is not helpful for understanding the logic of the computation, i.e., the mapping from input values to output values.

To understand the problem, consider the instruction $I_5 \equiv$ '`fact := fact * i`'. The variables `i` and `fact` have both been initialized to the value 1 at this point, so the value of the expression '`fact * i`' is inferred to be a constant. Constant propagation then simplifies this instruction to the assignment '`fact := 1`'. Arguably, this simplification does not preserve the logic of this computation because it suggests that this assignment computes a fixed constant value when, in reality, the value that is computed by this instruction depends on the number of iterations of the loop, which in turn depends on the input value. The same observation applies to the other arithmetic simplifications carried out on this trace. The problem arises because the simplification fails to take into account the fact that the instruction being simplified is control-dependent on the input-tainted instruction $I_4 \equiv$ '`if (i > n) goto Bot`', which induces an implicit information flow from the input to I_5.

We address this problem by restricting the propagation of constants across input-tainted conditional jumps. This is done as follows. We first identify control dependencies as described in Algorithm 1. Given an instruction X, let $ControlDeps(X)$ denote the set of input-tainted instructions in the execution trace that X is control-dependent on. Then, a backward-tainted arithmetic operation I is *simplifiable* only if every source operand of I is either an immediate operand, or else is defined by an instruction J such that $ControlDeps(J) = ControlDeps(I)$. Applying this condition to the trace of Figure 5(b), we find that instruction I_5 is control-dependent on the input-tainted instruction $I_4 \equiv$ '`if (i > n) goto Bot`', but its operands `fact` and `i`, which are defined by instructions I_3 and I_2 respectively, which are not control dependent on any instruction and therefore in particular are not control dependent on I_4. Thus, $ControlDeps(I_5) \neq ControlDeps(I_3)$ and so I_5 is not simplifiable. The constant value of `fact` defined by I_3 is therefore not propagated to I_5, which is what we want.

D. Control Flow Graph Construction

The final step in our deobfuscation process is to construct a CFG [24] from the simplified trace obtained from the trace simplification step. For deobfuscation purposes, one issue that arises in this context is that of reuse of code in a way that complicates the program's control flow structure. In obfuscated code, we very often find that a given functionality I—e.g., an emulator operation such as addition or subtrac-

```
     n := read()
     i := 1
     fact := 1
Top: if (i > n) goto Bot
     fact: = fact * i
     i := i + 1
     goto Top
Bot: write(fact)
     halt
```

(a) Static code

```
I1   n := read()
I2   i := 1
I3   fact := 1
I4   if (i > n) goto Bot
I5   fact := fact * i
I6   i := i + 1
I7   goto Top
I8   if (i > n) goto Bot
I9   fact := fact * i
I10  i := i + 1
I11  goto Top
I12  if (i > n) goto Bot
I13  fact := fact * i
I14  i := i + 1
I15  goto Top
I16  if (i > n) goto Bot
I17  write(fact)
I18  halt
```

(b) Unsimplified trace (input = 3). Input-tainted instructions are shown underlined.

```
I1   n := read()
I2   i := 1
I3   fact := 1
I4   if (i > n) goto Bot
I5   fact := ~~fact * i~~ 1
I6   i := ~~i + 1~~ 2
I7   goto Top
I8   if (i > n) goto Bot
I9   fact := ~~fact * i~~ 2
I10  i := ~~i + 1~~ 3
I11  goto Top
I12  if (i > n) goto Bot
I13  fact := ~~fact * i~~ 6
I14  i := ~~i + 1~~ 4
I15  goto Top
I16  if (i > n) goto Bot
I17  write(~~fact~~ 6)
I18  halt
```

(c) Result of oversimplification.

Fig. 5. An example illustrating over-simplification

Algorithm 2: Final Control Flow Graph Construction

Input: Set of simplified execution trace T
Result: Control flow graph G for T

Let B_0 be first basic block in T
$t_{curr} := v_{curr} := B_0$
$G := (V, E)$ where $V = \{v_{curr}\}$ and $E = \emptyset$
EdgeStk := NULL
while *there are unprocessed blocks in* T **do**
  let t_{next} be the next block after t_{curr} in T
  if t_{next} *is already a successor of* v_{curr} **then**
    $v_{next} := t_{next}$
  else if *a successor can be added to* v_{curr} **then**
    / add* t_{next} *as a successor to* v_{curr} **/*
    Let v_{next} be a basic block in G that its entry point has the same address as t_{next} in T
    if v_{next} = *NULL* **then**
      $v_{next} := t_{next}$
      add v_{next} to V
    add $e \equiv$ '$v_{curr} \rightarrow v_{next}$' to E
    push e on *EdgeStk*
  else
    / backtrack using EdgeStk */*
    pop $e \equiv$ '$a \rightarrow b$' from *EdgeStk*
    t_{curr} := block in T corresponding to a
    t_{next} := block in T corresponding to b
    v_{curr} := block in G corresponding to t_{curr}
    v_{next} := Duplicate(t_{next})
    add $e \equiv$ '$v_{curr} \rightarrow v_{next}$' to E
    push e on *EdgeStk*
  end
  $v_{curr} := v_{next}$
  $t_{curr} := t_{next}$
end
Output G

tion (in emulation-obfuscation), or a gadget for an operation such as copying one register to another (in return-oriented programming)—is implemented using a single code fragment C_I; control is then directed to C_I whenever the functionality I is needed in the program. This means that if there are k different occurrences of I in the original program, they will end up executing the same piece of code C_I in the emulated program k times, with k corresponding repetitions of C_I in the execution trace. A CFG constructed in a straightforward way will then have k pairs of control flow edges coming into and out of the code region C_I, which will cause the control flow behavior of the program to appear very tangled.

During deobfuscation, therefore, we try to construct the CFG in a way that attempts to untangle some of the paths by judiciously duplicating basic blocks. Intuitively, we want to minimize the amount of such code duplication, while at the same time reducing the number of "spurious" control flow paths (paths that are possible in the CFG constructed but which are not observed in the trace(s) used to construct the CFG). Solving this problem optimally seems combinatorially challenging, and related problems in computational learning theory that are known to be computationally hard: the problem of identifying a CFG that is consistent with a given trace (i.e., which admits that trace but may also admit other execution paths) can be modeled as that of constructing a DFA consistent with a given set of strings (i.e., which accepts those strings but may also accept other strings). Unfortunately the problem of finding the smallest DFA (or the smallest regular expression) that is consistent with a given regular language is NP-hard [25], [26] and is not even efficiently approximable [27].

Given these results, we augment the usual CFG construction algorithm [24] with heuristics aimed at balancing the number of vertices and the complexity of the constructed CFG, using a depth-first backtracking search to explore the search space as is shown in the Algorithm 2. We briefly sketch the algorithm here.

The simplified trace, from which we construct the deobfuscated control flow graph, is a sequence of instructions that can also be considered as a sequence of basic blocks such that if a block B is followed by a block B' in the (simplified) trace it corresponds to an edge $B \rightarrow B'$ in the corresponding control flow graph. Our algorithm traverses the sequence of basic blocks in the trace, constructing a control flow graph G using the usual CFG construction algorithm, by adding basic blocks and/or edges to G, as long as this does not violate any structural constraints of any vertex in G; currently, the primary structural constraint that is enforced is the *out-degree constraint*: namely, that a basic block ending with a conditional jump can have at most two successors or if it is ending with an indirect jump, there is no restriction on the number of its succesors. This requirement is checked at the line 9 of the Algorithm 2. If the algorithm encounters a situation where adding a block and/or edge to G would violate this structural constraint, it backtracks to the most recently added vertex that can be duplicated without violating the out-degree constraint (Algorithm 2 lines 18-25). This vertex is then duplicated, together with vertices and edges that were added to G more recently, after which the algorithm resumes in the forward direction.

Another problem that the simplification might cause is removing dynamically dead instrucitons that affects the final CFG in such a way that causes the CFG construction algorithm to produce a new basic block for the code in which dynamically dead instructions are missing. The final step of deobfuscation is to apply semantics-preserving transformations to simplify the control flow graph. In particular, we identify and merge basic blocks that differ solely due to dynamically dead instructions. The following snippet of code, to compute the factorial function, illustrates the problem:[2]

```
int factorial(int n) {
  int i, p;
  p = i = 1;
  while (n > 0) {
    p = p*i
    i = i+1
    n = n-1
  }
  return p;
}
```

Suppose this function is called with the argument `n = 2`. The resulting execution trace for this function is:

```
/* 1 */   i = 1
/* 2 */   p = 1
/* 3 */   n > 0?      /* n == 2 */
/* 4 */   p = p*i
/* 5 */   i = i+1
/* 6 */   n = n-1
/* 7 */   n > 0?      /* n == 1 */
/* 8 */   p = p*i
/* 9 */   i = i+1
/* 10 */  n = n-1
/* 11 */  n > 0?      /* n == 0 */
/* 12 */  return p
```

The statement at position 9 in this trace, '`i = i+1`', is dynamically dead, since the value it computes at that point in the execution is not used later, and so it is removed during trace simplification. When a control flow graph is constructed from the simplified trace, however, we get two different versions of the loop body:

```
p = p*i
i = i+1
n = n-1
n > 0?
```

and

```
p = p*i

n = n-1
n > 0?
```

The first of these corresponds to the iterations up to the last iteration, while the second corresponds to the last iteration. More generally, depending on the dependence structure/distance of the loop(s) we may get multiple such loop body fragments with some code simplified away. Such blocks are treated as distinct by the control flow graph construction algorithm, resulting in a graph that has more vertices, and is more cluttered, than necessary. A similar situation arises with function calls if some call sites use the return value but others do not.

We deal with this situation by identifying and merging basic blocks that are identical modulo dynamically dead instructions. Define two blocks B_1 and B_2 to be *mergeable* if the following conditions hold:

1) B_1 and B_2 span the same range of addresses (except possibly for any dynamically dead instructions at the beginning and/or end of either block).
2) [*Non-dynamically dead instructions*] If an instruction I occurs in both B_1 and B_2, then it is the identical instruction in both B_1 and B_2. I.e., the operands should not have changed (e.g. due to constant propagation).
3) [*Dynamically dead instructions*] For each instruction $I \in B_1$ that does not occur in B_2, I is dead if it is added into B_2 at the appropriate position; and analogously with instructions that are in B_2 but not in B_1.

To simplify the control flow graph, we repeatedly find mergeable basic blocks and merge them to obtain the final control flow graph.[3]

IV. Experimental Evaluation

We have evaluated our ideas using a prototype implementation of our approach. Execution traces of the original and obfuscated binaries were collected using a modified version of Ether [16]. Trace simplification was carried out on a machine with 2× quad-core 2.66 GHz Intel Xeon processors with 96 GB of RAM running Ubuntu Linux 12.04. The results of our experiments are discussed below. To quantify the similarity between the original and the deobfuscated programs (and, for completeness, the obfuscated programs as well), we use an

[2]In reality we work with assembly instructions. This example uses C code for the program, and a quasi-C syntax for the trace, for simplicity and ease of understanding.

[3]From an implementation perspective, it turns out to be simpler to modify the simplified trace to reintroduce, where necessary, dynamically dead instructions that had been simplified away, and then rebuild the control flow graph.

algorithm of Hu, Chiueh, and Shin for computing the edit distance between two control flow graphs [28]. Given two control flow graphs G_1 and G_2, this algorithm computes a correspondence between the vertices of G_1 and G_2 using maximum bipartite matching, then uses this correspondence to determine the number of edits, i.e., the number of vertex and edge insertion/deletion operations necessary to transform one graph to the other. To facilitate comparisons between CFGs of different sizes, we normalize the edit distance to the total size of the graphs being compared. Let $\delta(G_1, G_2)$ be the edit distance between two control flow graphs G_1 and G_2, then their similarity is computed as

$$sim(G_1, G_2) = 1 - \frac{\delta(G_1, G_2)}{|G_1| + |G_2|}$$

where $|G|$ is the size of the graph G and is given by the total number of vertices and edges in G. A similarity score of 0 means that the graphs are completely dissimilar; a similarity score of 1 means that the graphs are identical.

Our experimental samples, including source code for the test programs and executables for the original and obfuscated programs, are available at `www.cs.arizona.edu/projects/lynx/Samples/Obfuscated/`.

A. Emulation-based Obfuscation

We evaluated our deobfuscator using four commercial emulation-obfuscation tools: Code Virtualizer [1], EXECryptor [2], Themida [4], and VMProtect [3]. Code Virtualizer and VMProtect are representative of obfuscation tools that have been considered in previous work [5], [6]; these authors do not discuss EXECryptor so we do not know whether they are able to handle software obfuscated using this tool. As far as we know, none of the existing approaches on deobfuscation of emulation-obfuscated software are able to handle binaries obfuscated using Themida. When obfuscating programs using Themida, users can select various parameters, including the complexity of the VM instructions: for our experiments used the setting '*mutable CISC processor*' with one VM whose opcode type is '*metamorphic level-2*'.[4]

1) Single-Level Emulation: Single-level emulation refers to obfuscation where there is just a single level of emulation, namely, that of the emulator introduced by the obfuscation process. This is the only kind of emulation-based obfuscation considered thus far by other researchers on this topic.

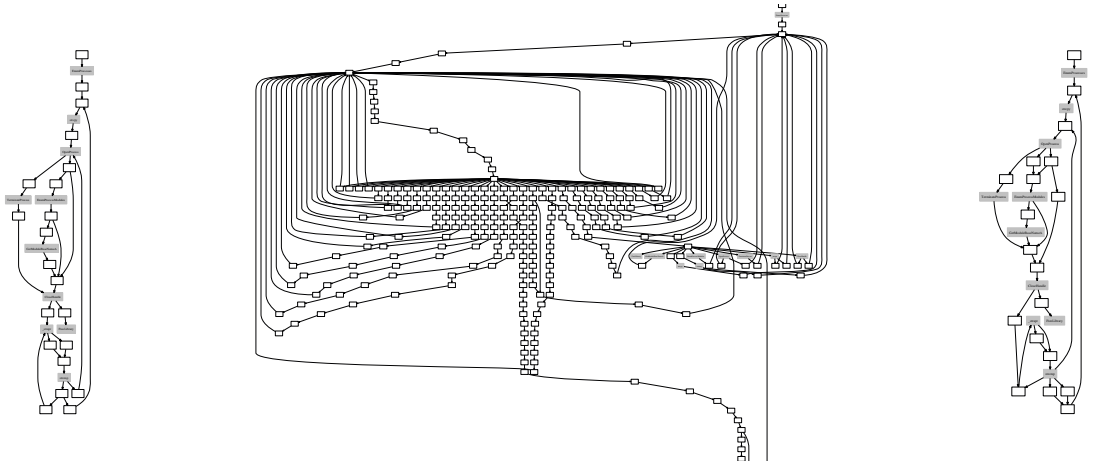
To evaluate the quality of deobfuscation results using our approach on single-level emulation, we applied the commercial obfuscators named above to several malware programs, whose source code we obtained from VX Heavens [31], together with two synthetic benchmarks we wrote ourselves. The malware programs we used were: *Blaster* [29], *Cairuh*, *epo*, *hunacha*, *newstar*, and *netsky_ae* [30]. Of these programs, *Blaster* is a network worm; *Cairuh* is a P2P worm; *hunatcha* is a file dropper; *newstar* and *epo* are file infectors that implement different file infection mechanisms to drop payloads into other files; and *netsky_ae* is a worm whose functionality we divided into different pieces: *netsky_ae1* searches and eliminates antivirus and monitoring software running on the system, *netsky_ae2* installs the malware for surviving the system boots, *netsky_ae3* infects the system with encrypted variations of the malware and *netsky_ae4* recursively copies the malware into shared folders. In addition to these malware programs, we used two synthetic benchmarks, *huffman*, and *matrix-multiply*, to explore how our techniques handled various combinations of conditionals and nested loops.

Space constraints preclude showing the full control flow graph of each of our test inputs; Figure 6 gives a high-level visual impression of the effect of emulation-based obfuscation, together with the deobfuscated programs obtained using our approach, for two different malware samples: *Netsky_ae1*, *Hunatcha*, and the *matrix multiply* program, that have reasonably interesting control flow structure, consisting of nested loops and conditionals; and three widely-used obfuscation tools: Code Virtualizer, ExeCryptor, and Themida. In order to focus the discussion on the core portion of the computation, the graphs shown omit the program setup/takedown and I/O code. It can be seen, from visual inspection, that the control flow graph resulting from deobfuscation is in each case very similar to that of the original program.

The results of the similarity comparisons are shown in Table I. Columns labeled '*Obf.*' give the similarity of the obfuscated programs with the original programs; those labeled '*Deobf.*' give the similarity between the deobfuscated programs and the original programs. Not surprisingly, the obfuscated programs are usually very different from the original code structurally: by and large these similarity numbers are in the 6%–8% range, with several programs showing similarities of less that 10%, and a few (e.g. *huffman*, *hunatcha* and *epo* for Code Virtualizer, and *huffman* for VMProtect) with similarity values over 15%. The exceptions here are *Cairuh*, *netsky_ae2* and *netsky_ae4* which because of having switch statements in their code, they are structurally similar to the virtualized binaries so they are in fact more similar to the obfuscated binaries than the other programs. By contrast, the control flow graphs resulting from our deobfuscation algorithm have significantly higher similarities. While nearly similar on average, they are highest for Code Virtualizer and Execryptor, ranging from 72% to 95% for Code Virtualizer and in the range of 75% to more than 94% for EXECryptor. On average the similarity values for Code Virtualizer and EXECryptor are 86.6% and 86.4%. The deobfuscation results are comparable for Themida and VMProtect, ranging from 82% to 96% for Themida and from 46% to 96% for VMProtect. However, it should be noted that our approach still achieves significant improvements in similarity relative to the obfuscated code.

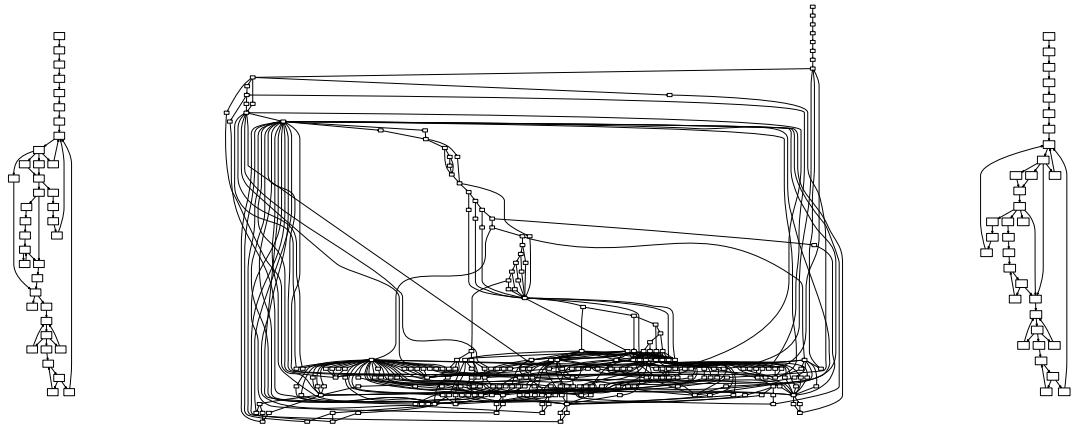
Our Ether-based tracing infrastructure crashed on the *Cairuh* and *blaster* programs obfuscated with Themida so we were unable to collect an execution trace for these programs.

[4] "Mutable CISC processor" and "metamorphic level-2" are settings in the Themida tool; the available documentation does not specify, in any further detail, exactly how these settings affect the low-level characteristics of the obfuscated code.



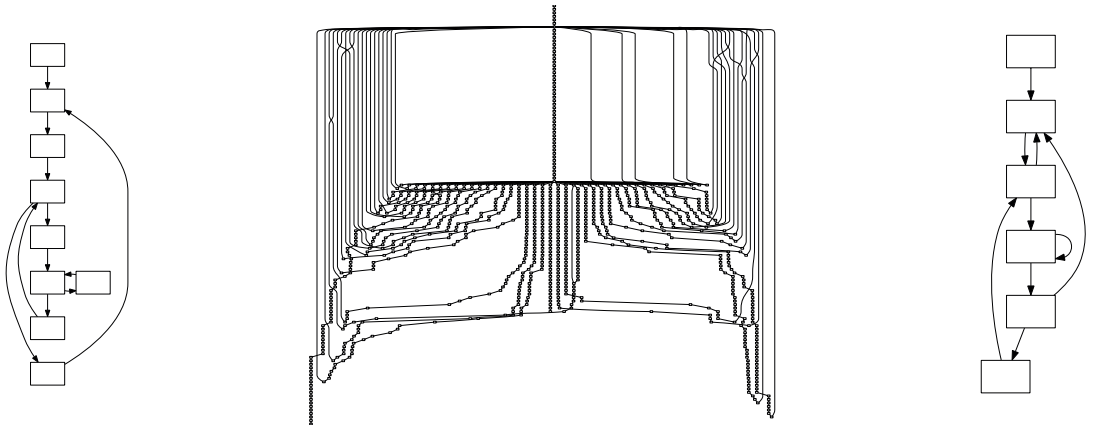
Original Obfuscated (cropped) Deobfuscated

(a) Netsky_ae1: Code Virtualizer



Original Obfuscated (cropped) Deobfuscated

(b) Hunatcha: ExeCryptor



Original Obfuscated (cropped) Deobfuscated

(c) Matrix multiply: Themida

Fig. 6. Effects of obfuscation and deobfuscation on the control flow graphs of some malware samples

PROGRAM	*Control flow graph similarity (%)*							
	CODE VIRTUALIZER		EXECRYPTOR		THEMIDA		VMPROTECT	
	Obf.	*Deobf.*	*Obf.*	*Deobf.*	*Obf.*	*Deobf.*	*Obf.*	*Deobf.*
huffman	20.75	72.24	06.08	83.50	06.03	83.91	16.45	46.40
hunatcha	22.43	90.30	04.82	90.04	05.60	84.84	15.57	73.65
matrix-mult	06.50	81.63	01.31	83.95	01.56	81.63	07.22	75.55
Cairuh	39.37	89.02	26.46	94.04	NA	NA	28.68	82.39
blaster	13.25	84.54	02.40	84.87	NA	NA	14.07	89.24
newstar	09.09	94.38	02.15	92.56	02.21	96.70	08.49	75.20
epo	29.26	92.51	07.86	80.92	09.28	81.23	20.03	96.28
netsky_ae1	19.78	88.03	08.19	87.27	06.15	84.14	19.00	82.81
netsky_ae2	50.90	80.85	13.12	93.40	19.75	88.17	24.50	89.95
netsky_ae3	11.52	92.85	02.43	85.49	03.84	82.81	09.35	94.36
netsky_ae4	30.30	86.60	20.71	75.04	14.04	82.66	22.65	87.85
AVERAGE	23.01	86.63	08.68	86.43	07.60	85.12	16.91	81.24

TABLE I
SIMILARITY OF ORIGINAL AND DEOBFUSCATED CONTROL FLOW GRAPHS: EMULATION-OBFUSCATION

In Figure 7 we have included the CFGs of a subtrace of the *netsky1_ae* program with instructions included in the graph: Figure 7(a) corresponds to the original program and (b) corresponds to the deobfuscated program obfuscated using Code Virtualizer. This shows that with the high level information that can be recovered by the CFGs, program semantic information is also included at the instructions level. For example in Figure 7, it can be seen that in both graphs, there is a test on the output of the `strcmp` function call marked with label 1. The program is trying to kill all the unwanted processes currently running in the system and by comparing process names with ones in a list, it determines whether to terminate the process or not. If the comparison satisfies, it calls `OpenProcess` (labeled with 2) and then terminates the process using a call to `TerminateProcess` (labeled with 3). There is correspondence between two graphs and the semantics are equivalent in both the original and deobfuscated programs. Getting this level of information from the obfuscated program, where the graph is shown on Figure 6(a), is very unlikely, if not impossible, and requires significant amount of time and efforts.

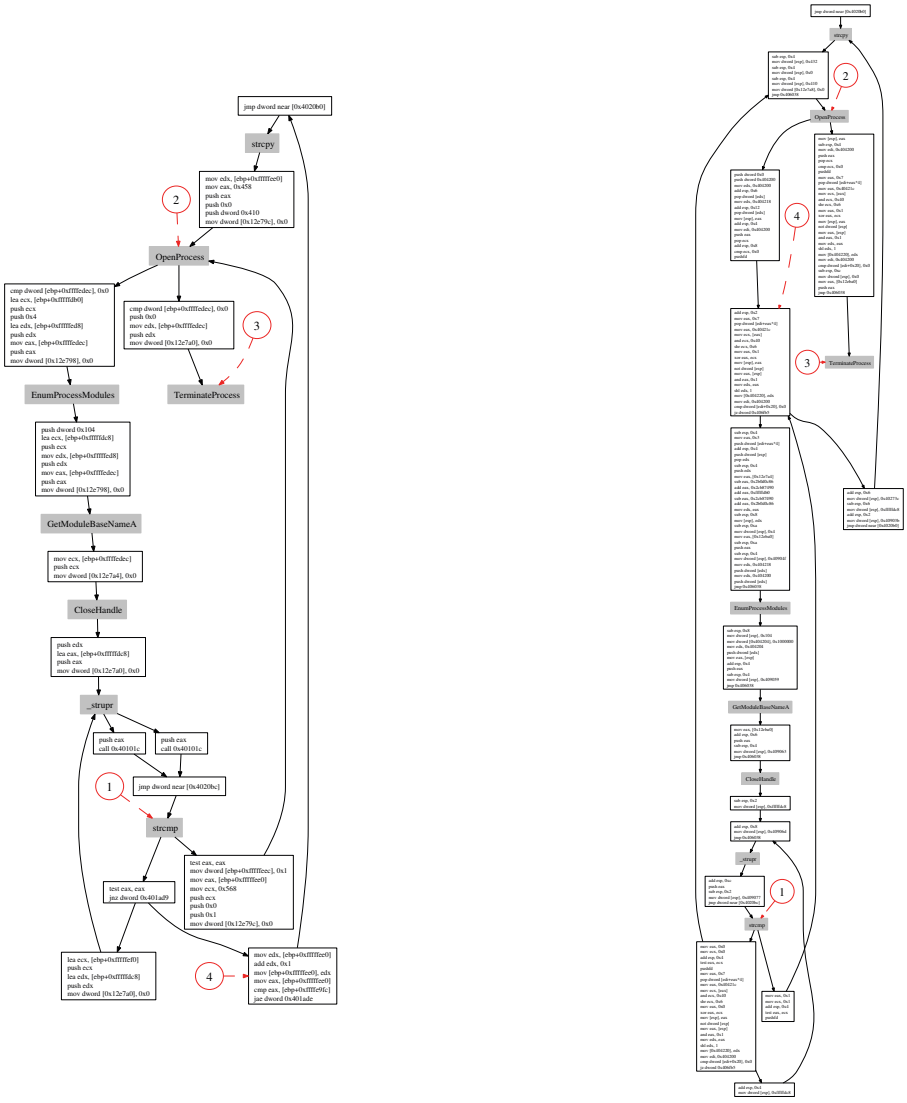
However, there is one difference between two graphs that should be noted here. As it was discussed in Section III-D, the CFG construction algorithm tries to balance between the code duplications and the number of paths in the final graph. Doing so, the CFG constructed for the deobfuscated program uses an existing block (pointed by label 4) rather than duplicating it for the corresponding block in original program (also pointed by label 4). This is mostly because in the original program, only one target branch is observed (for the basic block pointed by label 4) and so the CFG construction algorithm does not have a clue about the other branch existing in the original program. It should also be noted that this does not however affect the semantics of the program and the constructed graph still represents the original logic correctly and this is a general limitation for dynamic analysis where the code coverage is an issue rather a specific limitation of our approach.

Analysis speed depends partly on the input trace size but mostly on the number of iterations of code simplification needed, which in turn depends on how entangled the obfuscations are; there seems to be a non-linear component to the execution time that we are currently looking into. Execution times for the three largest trace files, *Cairuh*-VMProtect (6.4M instructions), *hunatcha*-Themida (7.7M instructions), and *huffman*-Themida (56.6 M instructions) are 188 sec, 244 sec, and 4,726 sec respectively, which translate to speeds of 34,042 instrs/sec, 31,557 instrs/sec, and 11,976 instrs/sec respectively.

We have also applied our deobfuscator to a number of emulation-obfuscated malicious binaries that we obtained from *virusshare.com*, including *Win32/Kryptik*, *Trojan-Downloader.Banload*, *Win32.Dubai*, *W32/Dialer*, and *Backdoor.Vanbot*. Space constraints preclude showing the original and simplified CFGs for these programs, so we briefly summarize our findings. We found that in the samples we tested, emulation was typically applied selectively to selected sensitive code regions, with multiple layers of unpacking added subsequently to further obfuscate the malicious payload. Our deobfuscator was able to remove all of the emulation and unpacking code, leaving only the logic of the malicious payload with a much simpler CFG. The time taken to perform this simplification for the malware samples we tested was around 10 minutes per sample.

Overall, these results show that while our prototype implementation is not yet perfect, it is nevertheless able to extract control flow graphs that closely resemble those of original unobfuscated programs. Notably, it is able to do this for both "ordinary" emulation-obfuscated programs and also Themida-obfuscated programs, which combine runtime unpacking with emulation and, as far as we know, are not handled by any previously proposed techniques for automatic deobfuscation. Considering that we make very few assumptions about the nature of the obfuscations applied, we consider this encouraging. We are currently working on improving our analyses to improve the deobfuscation results further.

2) Multi-level Emulation: We have also applied our approach to programs obfuscated using multiple levels of emulation, i.e., where one emulator interprets another emulator



(a) Original program

(b) Deobfuscated program

Fig. 7. Example of CFGs with instructions

which in turn interprets byte code for the program to be executed: the results are similar to those presented here, in that we are able to remove most of the obfuscation and recover deobfuscated control flow graphs that are very similar to those shown here. We selected a subset of our test programs which we used for single-level emulation, including *binary-search*, *bubble-sort* and *matrix-multiply* and obfuscated them using Code Virtualizer, and then applied another round of emulation using EXECryptor. Each of these programs therefore had two levels of emulation. We also wrote an emulator, modeled on DLXsim and SPIM, for a small RISC-like processor that we call *tinyRISC*, and ran it on hand-compiled byte-code for a binary-search program. This program was also obfuscated using CodeVirtualizer and EXECryptor and is included as *tinyRISC:bin-search*; this program uses three levels of emulation (the tinyRISC emulator, Code Virtualizer, and EXECryptor). Table II shows the similarity numbers for the obfuscated and deobfuscated CFGs of our test programs. It can be seen that the similarity of the deobfuscated CFGs and the original CFGs ranges from 80.6% to 87.9%. This shows that our approach is effective in cutting through multiple levels of emulation.

The similarity between the numbers for the multi-level emulated binaries and the ones obfuscated using only Code Virtualizer in Table I suggests that applying additional levels of emulation does not change the structure of the underlying interpreted program, although the obfuscated programs are quite different (see CFG similarity numbers for the obfus-

cated programs in the two cases), and the execution traces differ significantly with those for multi-level emulation being significantly larger.

PROGRAM	*No. of Levels*	*CFG similarity (%)*	
		Obf.	*Deobf.*
binary-search	2	4.45	85.29
bubble-sort	2	6.41	80.64
matrix-multiply	2	5.26	81.63
tinyRISC:bin-search	3	4.45	87.87
AVERAGE		5.14	83.85

TABLE II
SIMILARITY OF ORIGINAL AND DEOBFUSCATED CONTROL FLOW GRAPHS: MULTI-LEVEL EMULATION. *No. of Levels* GIVES THE NUMBER OF EMULATION LEVELS IN THE OBFUSCATED CODE.

B. Return-Oriented Programs

We evaluated our prototype implementation with two different sets of ROP test cases. The first set of binaries were simple synthetic programs including *factorial*, *fibonacci*, *matrix-multiply* and *bubble-sort*. These programs were implemented by chaining relevant ROP gadgets from Windows system libraries such as `ntdll.dll` and `msvcrt.dll` rather than a high level programming language to carry out the intended computation so they can simulate the behavior of ROP attacks. We chose these programs because they have enough complex structures such as loops and conditional statements to measure the ability of a reverse engineering system which tries to recover the logic of the underlying computation. For comparison purposes we also created the non-ROP version of the programs which are written in C. We also applied our approach to several ROP malware samples, but found that our ROP malware samples had a relatively simple control flow structure since all they were trying to do was to change the access permissions on some memory pages to make them executable. As a result, our hand-crafted ROP benchmarks presented a greater challenge for deobfuscation than the malware samples we tested. For our hand-crafted ROP sample, we tried to use ROPC [31] to create the ROP programs but, for a variety of technical reasons, were not able to get it to work.

The similarity numbers for our synthetic programs are presented in Table III. The column labeled *Obf.* shows the CFG similarity of the ROP version of the program to its non-ROP version and column labeled *Deobf.* shows the similarity of the deobfuscated ROP program to its non-ROP version. The table shows that our method is also able to reverse engineer the ROP gadgets and produce a very similar control flow graph to the non-ROP version by simplifying the ROP version execution trace.

PROGRAM	*CFG similarity (%)*	
	Obf.	*Deobf.*
factorial	47.61	88.88
fibonacci	30.61	85.71
matrix-multiply	64.51	79.22
bubble-sort	48.22	82.85
AVERAGE	47.73	84.16

TABLE III
SIMILARITY OF ORIGINAL AND DEOBFUSCATED CONTROL FLOW GRAPHS: ROPS

We have included the set of control flow graphs of two ROP programs, *factorial* and *fibonacci* in Figure 8 very similar to Figure 6. Note that the factorial program has a nested loop; the reason is that we did not find a multiplication gadget in `ntdll.dll` or `msvcrt.dll`, so we simulated this using a loop of additions.[5]

C. Comparison With Coogan et al.

We tested our approach against that of Coogan *et al.* [6]; the results are shown in Figure 9. Coogan's approach results in complex equations that are difficult to map to CFGs, especially for nontrivial programs. Our approach, by contrast, produces CFGs that can be meaningfully compared to the original program's CFGs. So we think that our approach produces more understandable results than Coogan's. We ran Coogan's tool on their set of test programs and mapped the resulting *relevant subtraces* (which is equivalent to the *deobfuscated program* in our terminology) to CFGs. We first applied our tool on the traces used by Coogan *et al.* in their experiments [6] and compared the similarity of the resulting deobfuscated traces with the original ones. To compare the result of the two tools, we also generated CFGs of the relevant subtraces produced by their tool and compared the CFGs to the original programs. It can be seen, from Figure 9, that our system outperforms Coogan's tool with a 30% to 60% higher similarity numbers in all the programs. We were not able to get a result of their tool on the *md5* program obfuscated using Code Virtualizer because the computation did not finish on time so we did not have any data for that. The small difference between similarity numbers of the programs that are common in our set of input programs and the set they used for evaluation, e.g., *hunatcha*, is that the programs used by Coogan *et al.* and represented in Figure 9 are slightly different from those used for Table I.

Coogan *et al.* do not apply their technique to obfuscations other than emulation, nor do they provide results for multi-level emulation.

V. DISCUSSION

Like all other work on automatic malware analysis, we presuppose that the malicious code has been analyzed and (since we are using dynamic analysis) an execution trace has been collected. If a program attempts to thwart analysis via anti-analysis defenses then those defenses will have to be overcome before our techniques can be applied. This problem is common to all work on automated malware analysis and is orthogonal to the topic of this paper, so we do not pursue it further here.

[5]This problem with unavailability of multiplication gadgets in Windows system libraries, and a solution using iterated addition, is also discussed by Roemer *et al.* [7].

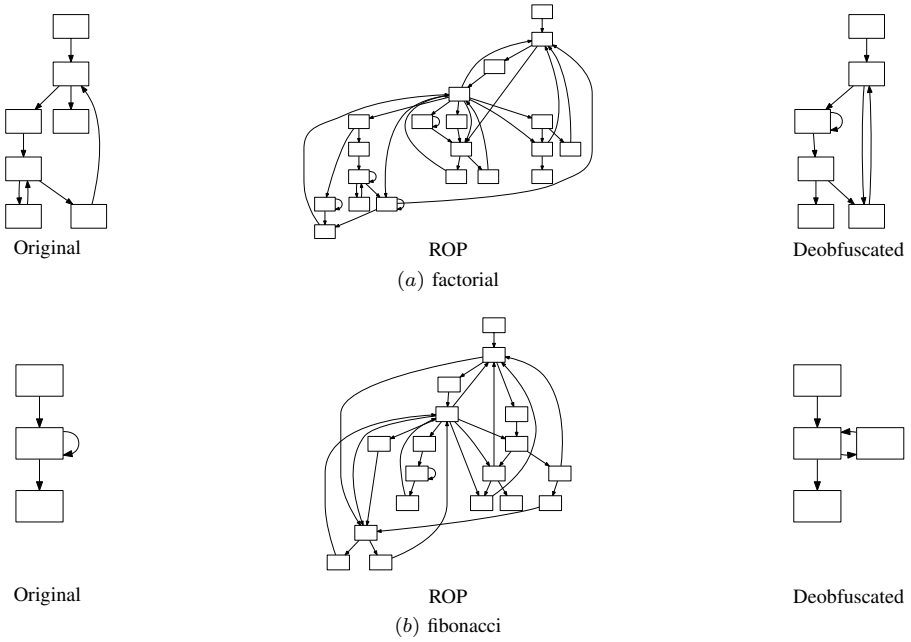


Fig. 8. Some examples of ROP deobfuscation results

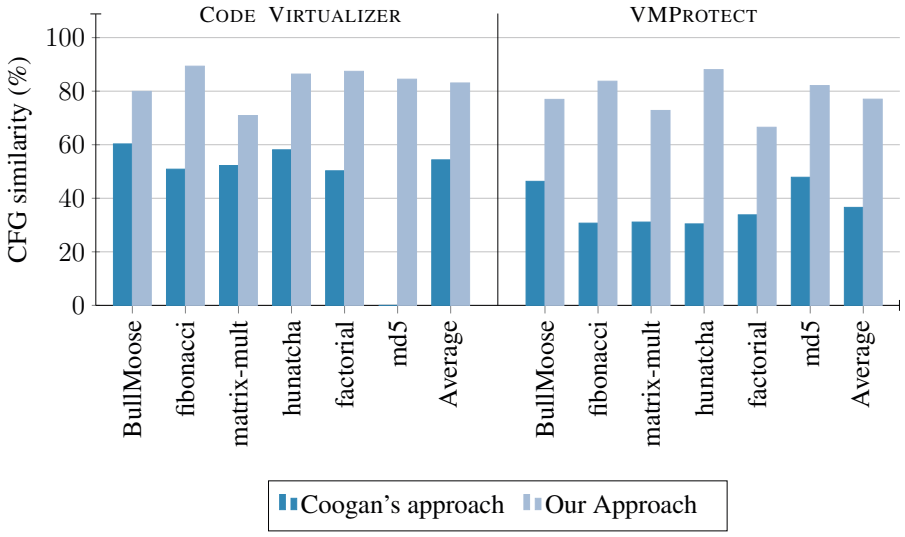
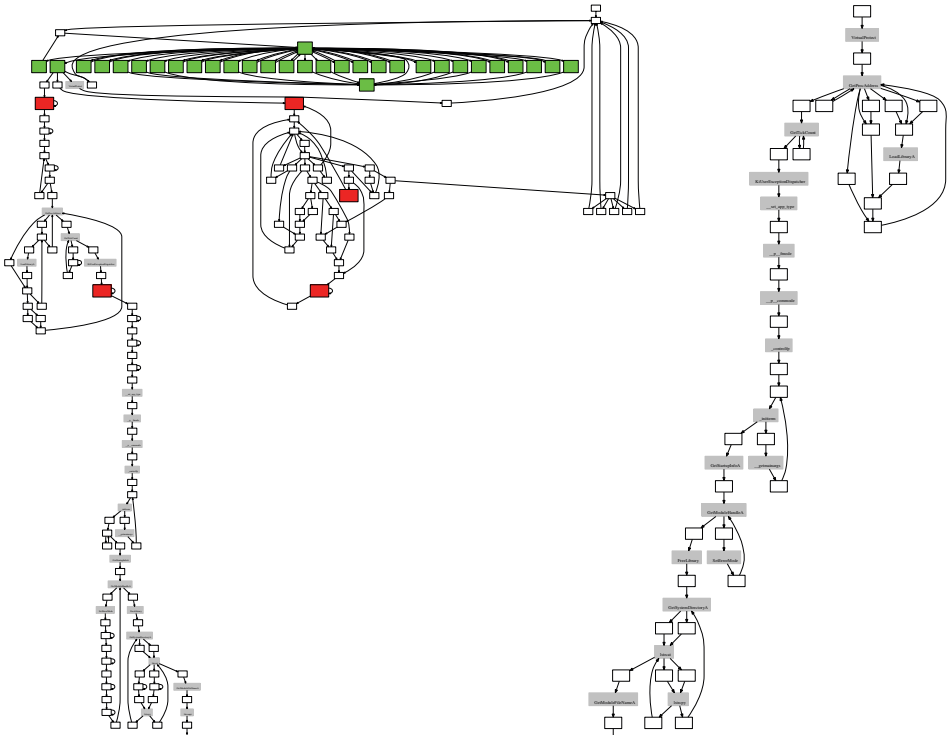


Fig. 9. Comparison with Coogan *et al.*

Code coverage can be an issue since we rely on dynamic analysis, where only one execution path through the program is observed. To overcome this problem we apply multi-path exploration techniques based on concolic execution to identify inputs that will exercise alternative execution paths and increase code coverage [21], [18]. The constraints used to identify such alternative inputs are computed from an execution trace; in our system one can use either the original (obfuscated) trace or the simplified trace for this. Our experiments indicate that, due to the effects of obfuscation, the original traces are often much larger and more complex than the simplified traces, and result in correspondingly larger and more complex constraints whose solutions require more time and memory. We found that, in many cases, the constraint solver (our experiments used STP [32]) fails to find a solution for constraints obtained from the original traces, e.g., because it runs out of time or memory, but is able to solve those obtained from the simplified traces. The process of deobfuscation is therefore also helpful for exploring alternative behaviors in obfuscated executables.



Original (cropped) Deobfuscated (cropped)

Fig. 10. Partial Control flow graphs for *Win32/Kryptik.OHY* Trojan before and after deobfuscation. The emulation-obfuscated portion of the program is shown in green while basic blocks that perform code unpacking, i.e., write to memory locations that are subsequently executed as code, are shown in red.

As discussed in Section II-C, our threat model assumes that the adversary knows our semantics-based approach to deobfuscation. We recognize three ways in which an adversary can try to reduce the effectiveness of our analysis. The first is to leave the I/O operations of the input program unchanged but entwine the obfuscation code with the original input-to-output computation much more deeply in order to prevent the obfuscation code from being simplified away. The second is to introduce additional input/output operations into the program along with (obfuscation) code that operates on the new input-to-output flow of values that this gives rise to. The third approach is to hide some of the computation performed by the program.

With the first approach, entwining the obfuscation code with the input-to-output flow of values of the original program in a way that is semantically significant, but which at the same time can be guaranteed to preserve the behavior of the program being obfuscated, is a challenging problem in general. The reason is that even simple transformations can affect the observable behavior of the program, e.g., by changing use/definition relationships, introducing arithmetic overflow/underflow, or perturbing condition code settings. This means that any such entanglement of the obfuscation code will, at the very least, require sophisticated program analyses that go well beyond the capabilities of today's obfuscation tools. Alternatively, instead of relying on general-purpose tools capable of obfuscating arbitrary programs, the adversary could try to hand-craft custom malware where the obfuscation code is semantically integrated into the program logic. While such an approach would reduce the efficacy of our approach, it would also require a lot more time and effort for malware writers and would not scale.

The second approach actually changes the program's semantics (i.e., its observable interactions with its environment). Since deobfuscation must preserve program semantics, a deobfuscation tool cannot reasonably be expected to automatically disregard some of the semantically significant operations of the program. Thus, our approach will not be able to automatically recover the logic of the original program in this case. However, our ideas can be easily extended to deal with such obfuscations interactively: the tool user can (optionally) specify some set of input and/or output operations to be disregarded, and the deobfuscation tool can simply not perform taint propagation for the disregarded operations.

With the third approach, some of the logic of computation can be hidden by performing the computation elsewhere, e.g.,

on a remote host, where it cannot be observed. We note that this would be a problem for every approach to automatic deobfuscation that we are aware of, and believe that it is a fundamental limitation of any automatic deobfuscation tool.

In summary, for each of these cases we significantly raise the bar for obfuscation of malicious code.

VI. Related Work

The work that is philosophically closest to ours is that of Coogan *et al.* [6], who use equational reasoning about assembly-level instruction semantics to simplify away obfuscation code from execution traces of emulation-obfuscated programs. While their goals are similar to ours, the technical details are very different. The biggest difference between the two is in the processing and simplification of execution traces. The equational reasoning approach of Coogan *et al.* has some significant drawbacks, the most important being that it is difficult to control the equational simplification, making it hard to separate out the different components of nested loops or complex control flow. This makes it difficult for their approach to extract the logic of the underlying computation into higher-level structures such as control flow graphs or syntax trees. By contrast, our approach offers a lot more control over the deobfuscation process and allows us to recover higher-level representations, such as control flow graphs, with a high degree of precision, as illustrated by the data in Figure 9. Importantly, Coogan *et al.* limit themselves to emulation-based obfuscation, and provide data only for one level of emulation; by contrast, we are able to handle multiple levels of emulation with good results, and applies to other kinds of programs, e.g., ROPs.

Sharif *et al.* describe an approach [5] that works from the outside in: it first reverse engineers the VM emulator; uses this information to work out individual byte code instructions; and finally, recovers the logic embedded in the byte code program. This outside-in approach can be very effective when the structure of the emulator meets the assumptions of the analyzer. However, when the emulator uses techniques that do not fit these assumptions the deobfuscator may not work well. For example, this approach does not fully deobfuscate code that has been obfuscated using Themida, which virtualizes the unpacker routine for emulator instructions; for such programs, it is able to automatically recover only the unpacker logic (rather than that of the application), with further analysis then done manually. We have recently seen similar characteristics in code obfuscated with other emulation-based obfuscators as well: e.g., a malware sample for *Win32/Kryptik*, whose executable we obtained from *virusshare.com*, was found to have been obfuscated using Code Virtualizer, with emulation-obfuscation applied to just the top-level unpacker routine rather than the application logic (see Figure 10). We conjecture that this selective application of emulation-based obfuscation may have been motivated by a desire to avoid the space and time overheads that would result from applying this obfuscation to the entirety of the code; nevertheless, this development suggests that obfuscation-specific approaches that focus on identifying and reverse-engineering the emulator may become less effective in the face of selective application of obfuscation. This approach may also not generalize easily to code that uses multiple layers of emulation, since it may be difficult to distinguish between instruction fetches for various emulators.

Some researchers have proposed static approaches for simplifying (quasi-)interpretive code. Udupa *et al.* [33] discuss techniques for deobfuscating code that has been obfuscated using *control flow flattening* [34], which in some ways resembles emulation-based obfuscation. Jones *et al.* [35] describe a technique called *partial evaluation* for specializing away interpretive code. The analyses and transformations described in these works are static, which suggests that it may not be straightforward to apply them to highly obfuscated malware binaries, e.g., due to dynamic unpacking and self-modifying code.

There is a significant and growing body of literature on return-oriented programming, but most of it deals with attacks [7], [8], [10], [11] or defenses [9], [36]–[38]. Lu *et al.* discuss the conversion of ROP shellcode to semantically equivalent shellcode that does not use ROP [9], but this work is specific to ROP and not a generic technique.

VII. Conclusions

This paper describes a generic approach to deobfuscation of executable code. Instead of making strong assumptions about the obfuscation, e.g., the structure of the emulator, we consider the semantics of the program in terms of the input-to-output transformation it implements, and focus on identifying, extracting, and simplifying the code that carries out this transformation. We have evaluated our approach on emulation-based obfuscation and return-oriented programs. Experiments using sophisticated commercial obfuscation tools indicate that our approach is effective in stripping out the obfuscation and extracting the logic of the original code.

VIII. Acknowledgments

We are grateful to Patrick Chan for his implementation of the control flow graph similarity algorithm used in our evaluation.

References

[1] Oreans Technologies, "Code virtualizer: Total obfuscation against reverse engineering," `www.oreans.com/codevirtualizer.php`.

[2] StrongBit Technology, "EXECryptor – bulletproof software protection," `www.strongbit.com/execryptor.asp`.

[3] VMProtect Software, "VMProtect – New-generation software protection," `www.vmprotect.ru/`.

[4] Oreans Technologies, "Themida: Advanced windows software protection system," `www.oreans.com/themida.php`.

[5] 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.

[6] K. Coogan, G. Lu, and S. Debray, "Deobfuscating virtualization-obfuscated software: A semantics-based approach," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011, pp. 275–284.

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 2:1–2:??, Mar. 2012.

[8] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *proc. ACM Conference on Computer and Communications Security*, 2007, pp. 552–561.

[9] K. Lu, D. Zou, W. Wen, and D. Gao, "deRop: removing return-oriented programming from malware," in *Proc. 27th. Annual Computer Security Applications Conference (ACSAC)*, Dec. 2011, pp. 363–372.

[10] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, 2011, pp. 30–40.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. ACM Conference on Computer and Communications Security*, 2010, pp. 559–572.

[12] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. 13th USENIX Security Symposium*, Aug. 2004.

[13] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, Nov. 2007.

[14] J. Stoy, *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT, 1977.

[15] P. Ferrie, "Prophet and loss," *Virus Bulletin*, Sep. 2008, `www.virusbtn.com/virusbulletin/archive/2008/09/vb200809-prophet-loss`.

[16] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008, pp. 51–62.

[17] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 255–264.

[18] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[19] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: A platform for in-vivo multi-path analysis of software systems*. ACM, 2011, vol. 39, no. 1.

[20] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*. ACM, 2005, vol. 30, no. 5.

[21] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 231–245.

[22] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[23] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th. ACM Symp. Principles of Programming Languages (POPL 1998)*, Jan. 1998, pp. 184–196.

[24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers – Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley, 2007.

[25] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, vol. 39, no. 3, pp. 337–350, 1978.

[26] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.

[27] L. Pitt and M. K. Warmuth, "The minimum consistent DFA problem cannot be approximated within any polynomial," *J. ACM*, vol. 40, no. 1, pp. 95–142, 1993.

[28] X. Hu, T.-C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. ACM Conference on Computer and Communications Security*, Nov. 2009, pp. 611–620.

[29] W. worm, `www.cert.org/historical/advisories/CA-2003-20.cfm`.

[30] W32.Netsky.AE, `www.symantec.com/security_response/writeup.jsp?docid=2004-102522-4640-99&tabid=2`.

[31] pakt, "ROPC – Turing complete ROP compiler," `http://gdtr.wordpress.com/2013/12/13/ropc-turing-complete-rop-compiler-part-1/`.

[32] V. Ganesh and T. Hansen, "STP," `https://github.com/stp/stp`.

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

[34] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proc. International Conference of Dependable Systems and Networks*, Jul. 2001.

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

[36] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: a detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Mar. 2011, pp. 40–51.

[37] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proc. 26th. Annual Computer Security Applications Conference (ACSAC)*, Dec. 2010, pp. 49–58.

[38] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.