Symbolic Execution of Obfuscated Code

Babak Yadegari, Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{babaky, debray}@cs.arizona.edu

ABSTRACT

Symbolic and concolic execution find important applications in a number of security-related program analyses, including analysis of malicious code. However, malicious code tend to very often be obfuscated, and current concolic analysis techniques have trouble dealing with some of these obfuscations, leading to imprecision and/or excessive resource usage. This paper discusses three such obfuscations: two of these are already found in obfuscation tools used by malware, while the third is a simple variation on an existing obfuscation technique. We show empirically that existing symbolic analyses are not robust against such obfuscations, and propose ways in which the problems can be mitigated using a combination of fine-grained bit-level taint analysis and architecture-aware constraint generations. Experimental results indicate that our approach is effective in allowing symbolic and concolic execution to handle such obfuscations.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: [Software/Program Verification]

Keywords

Symbolic Execution; Obfuscation; Reverse Engineering; Taint Analysis

1. INTRODUCTION

Symbolic and concolic execution play important roles in a variety of security and software testing applications, e.g., test case and exploit generation [4, 5, 9, 17, 34], vulnerability detection [5, 6, 10], and code coverage improvement in dynamic analysis of malware code [2, 3, 26]. The general idea behind symbolic/concolic execution is to represent computations along a particular execution path using logical formulas and apply constraint solving techniques to identify inputs that would cause the program to take alternative execution paths. Analyses based on symbolic execution are especially important for dealing with programs that are difficult to analyze using conventional techniques. This makes the precision of such analyses an important consideration in security applications: on the one hand, identifying too many candidate execution paths, with corresponding inputs, can overwhelm the analysis and slow down processing; on the other hand, missing some execution paths can cause the analysis to fail to explore important parts of the input program.

Given the importance of symbolic analysis for code coverage improvement in dynamic analysis of potentially malicious code, it is important to identify and understand any potential weaknesses of this approach. Previous studies have discussed attacks on symbolic execution systems using cryptographic hash functions [36] or unsolved mathematical conjectures [43] to construct computations that are difficult to invert. These are sophisticated attacks and help define theoretical boundaries for symbolic analyses, however they do not speak to potential problems in symbolic analysis arising out of code obfuscation techniques used by existing malware.

It turns out that several existing code obfuscation techniques used by malware (or simple variations on them) can significantly affect the precision of current concolic analyses. For example, some obfuscations, such as those used in the software protection tool EXECryptor [38], can cause large amounts of overtainting and lead to a path explosion in the symbolic analysis; others, such as those used by the obfuscation tool VMProtect [41], transform conditional branch instructions into indirect jumps that symbolic analyses find difficult to analyze; and finally, a form of runtime code self-modification, variations of which we have seen in existing malware, can conceal conditional jumps on symbolic values such that they are not detected by concolic analysis. This situation is problematic because a significant motivation behind using symbolic/concolic execution in malware analysis is to get around code obfuscations. This makes it especially important to devise ways to mitigate such loss of precision when performing symbolic analysis of obfuscated code. This paper takes a first step in this direction.

This paper makes two contributions. First, it identifies shortcomings in existing concolic analysis algorithms by describing three different anti-analysis obfuscations that cause problems for symbolic execution. These obfuscations were selected because (1) they, or simple variants of them, are currently already used in malware, e.g., through tools like VMProtect and EXECryptor; and (2) the problems they cause for symbolic execution are not discussed in the research literature. Second, we describe a general approach, based on a combination of fine-grained taint analysis and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813663.

architecture-aware constraint generation, that can be used to mitigate the effects of these obfuscations. For the sake of concreteness, the discussion is in many places formulated in terms of the widely used x86 architecture; however, the concepts are general and apply to other architectures as well. Our experiments indicate that the approach we describe can significantly improve the results of symbolic execution on obfuscated programs.

The rest of the paper is organized as follows: Section 2 discusses background on concolic execution and introduces problems that arise in concolic analysis of obfuscated code. Section 3 discusses these challenges in greater detail. Section 4 describes our approach for dealing with these challenges. Section 5 presents experimental results from evaluation of a prototype implementation of our approach. Section 6 discusses related work, and Section 7 concludes.

2. BACKGROUND

2.1 Concolic Execution and Input Generation

Concolic (concrete+symbolic) execution uses a combination of concrete and symbolic execution to analyze how input values flow through a program as it executes, and uses this analysis to identify other inputs that can result in alternative execution behaviors [17, 34]. The process begins with certain variables/locations—typically, those associated with (possibly a subset of) the program's inputs—being marked as "symbolic." The instructions of the program are then processed as follows: if any of the operands of the instruction are marked symbolic, then the instruction is "executed" symbolically: the output operands of the instruction are marked as symbolic, and the relationship between the input and output operands of the instruction is represented as a constraint between the corresponding symbolic variables; otherwise, the instruction is executed normally and the program's state is updated. If a location or variable x becomes marked as symbolic, we say that x "becomes symbolic." The constraints collected along an execution path characterize the computation along that path in terms of the original symbolic variables, and can be used to reason about what inputs to the program can cause which branches in the program to be taken or not. Symbolic analysis can identify input classes to the program if there are control transfers in the program affected by the input values [22] by which program takes different execution paths.

2.2 Concolic Execution of Obfuscated Code

Figure 1 shows the problem with concolic analysis of obfuscated code. Our test program, shown in Figure 1(a), consists essentially of a single symbolic variable and two `if` statements, nested one inside the other, that give rise to a total of three distinct execution paths. Our goal is to use concolic execution to identify different inputs that will, between them, cover all three execution paths. Symbolic execution of this simple program is almost trivial: the concolic execution engine S2E [10] finds just two states and makes just seven queries, and the analysis takes less than 20 seconds overall. If we run this simple program through the obfuscation tool VMProtect [41], however, the results are dramatically different: a depth-first search strategy times out after more than 12 hours, having encountered close to 15,000 states and generated over 14,000 queries, but failing to generate any alternative inputs. A random search strategy does somewhat

```
int main(int argc, char **argv){
    int n = atoi(argv[1]);  /* n is symbolic */
    int retVal;
    int r = n+6;

    if(r < 10){
        retVal = 10;
        if (r == 6){
            retVal = 4;
        }
    } else {
        retVal = 12;
    }
    printf("%d\n", retVal);
    return retVal;
}
```

(a) Program source code (unobfuscated)

Search strategy	*Version*	*No. of states*	*No. of queries*	*Analysis time (sec)*
DFS	original	2	7	18
	obfuscated	14,928	14,015	time out (> 12 hrs)
Random	original	2	7	17
	obfuscated	25,800	25,094	14,160

(b) Analysis statistics (S2E)

Figure 1: Effects of code obfuscation on concolic analysis performance (Obfuscator: VMProtect [41]; concolic engine: S2E [10])

better in that it does not time out, but it takes nearly 800 times as long to generate alternative inputs compared to the unobfuscated version. This strategy encounters 25,800 states and generates more than 25,000 queries—an increase of four orders of magnitude. That such a trivial program should pose such a formidable challenge to symbolic execution when it has been obfuscated is sobering in its implications for more complex code: VMProtect and other similar obfuscators have been used for protecting malware against analysis for a decade or more (e.g., the Ilomo/Clampi botnet, which used VMProtect to protect its executables, was encountered in 2005 [14]). The problem is not specific to S2E: for example, when invoked on the obfuscated version of the program shown above, Vine [37] exits with an error message. The remainder of this paper examines the reasons underlying the problems described above and some possible ways by which the problems may be mitigated or remedied.

3. ANTI-CONCOLIC OBFUSCATIONS

While there has been a great deal of work on constructing and defeating different kinds of obfuscations, for the purposes of this paper we are concerned primarily with obfuscations that affect concolic analysis, focusing in particular on concolic analysis to improve code coverage in obfuscated and malicious code.[1] Such analyses use constraints on exe-

[1]The obfuscation tools we used to evaluate our techniques, discussed in Section 5, incorporate many additional obfuscations, but in our experience these other obfuscations did not have much of an effect on symbolic execution.

				Overflow				Sign	Zero				Parity		Carry
Flag		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF
Bit position	15	14	13-12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2: x86 `FLAGS` register [20]

cution paths leading up to conditional jumps to determine alternative inputs that can cause a different execution path to be taken. There are basically two broad ways in which this approach can be attacked:

1. The conditional jump can be manipulated in ways that make it difficult to identify a relationship with the original inputs:
   (a) The conditional jump can be transformed into an indirect jump whose target depends on the predicate of the original conditional jump.
   (b) The conditional jump can be transformed into a different conditional jump whose predicate depends on, but is different from, that of the original conditional jump.
2. The conditional jump, or its relationship with the input, can be concealed:
   (a) The conditional jump can be injected into the instruction stream at runtime, in the form of a direct unconditional jump, using conditional code modification ("symbolic code").
   (b) Implicit information flows can be used to conceal a conditional jump's dependence on inputs.

Of the possibilities listed above, this paper focuses on approaches 1(a), 1(b), and 2(a). The use of implicit flows (item 2(b) above) has been discussed elsewhere by Cavallaro *et al.* [8]. Sharif *et al.* discuss using cryptographic hash functions to realize an extreme form of approach 2(b) [36]; the discussion here considers simpler (and stealthier) forms of this approach that can nevertheless pose problems for concolic analysis. We have observed these obfuscations, or simple variants of them, in existing malware.

3.1 Conditional Jump to Indirect Jump Transformation

In the x86 architecture, conditional logic of the form

if e **then** S

is usually realized as follows: first, the expression e is evaluated and the condition code flags set; then, depending on the predicate involved in e, the appropriate combination of flags is used in a conditional branch instruction:

```
FLAGS := evaluate $e$
j$cc$ $A_S$
```

where cc represents the particular combination of flags corresponding to the predicate in e, and A_S is the address of the code for S. The architecture of `FLAGS` register on x86 processors is shown in Figure 2. However, this same effect can be realized by using the condition code flags resulting from the evaluation of e to compute the target address:

```
FLAGS := evaluate $e$
$r$ := $f$(FLAGS)      /* compute target address */
jmp $r$
```

In this case, the function f uses the condition code flag values to compute the target address; in particular, when the flag values indicate that the predicate e is true, the address computed by f(`FLAGS`) is A_S. A key difference between these two approaches is that in the first case, the use of a conditional branch instruction makes explicit the two possible control flow targets that are possible. This is not the case, however, for the indirect jump in the second case. As a result, the indirect jump is harder to analyze symbolically than the first: Schwartz *et al.* refer to this as the *symbolic jump problem* [33].

Obfuscators sometimes exploit this situation by transforming conditional branches to indirect jumps. This is illusrated by the following example.

Example 3.1. Consider the following code fragment:

```
1   $r_0$ := input();
2   FLAGS := test($r_0$)   /* x86:  test */
3   push(FLAGS)          /* x86:  pushf */
4   $r_1$ := pop()
5   $r_2$ := and $r_1$, 0x40
6   $r_3$ := 0x500000
7   $r_4$ := or $r_3$, $r_2$
8   jmp $r_4$
```

Instructions 2–4 above check the input value and move the condition code flags into register r_1. After some bit manipulation (instruction 5), it is bitwise or'd with the value in register r_3 (instruction 7). The resulting value is then used as the target of an indirect jump (instruction 8).

What is actually going on here is that instruction 5 extracts the bit corresponding to the Zero Flag (`ZF`), in bit position 6, from r_1 into r_2. The result of the bitwise or operation (instruction 7) is therefore either `0x500040` (if `ZF` had the value 1 after instruction 2) or `500000` (if `ZF` was 0). The indirect jump at instruction 8 is therefore really a conditional jump to one of these two addresses depending on the value of `ZF` from instruction 2. ▮

While this example is couched in terms of the widely-used x86 architecture, the ideas are not x86-specific: e.g., the ARM architecture allows similar direct manipulation of condition code bits with its `MSR`/`MRS` instructions. Such obfuscations are particularly an issue with *virtualization-based obfuscation*, where the program being obfuscated is translated into a byte-code like representation of the instruction set of a custom virtual machine (VM) and interpreted using a custom interpreter for that VM. Several commercial software protection tools are based on this approach [30, 31, 38, 41]; these tools are also used sometimes to protect malware code [16, 39]. While the details of the interpreters differ from one obfuscation tool to another, their high-level structure is typically that of a conventional fetch-dispatch-execute loop. The VM has its own *virtual instruction pointer* (VIP), which it uses to access the byte-code instructions it interprets. The VIP is initialized to the program's entry point in the byte-code, and its subsequent values are controlled by the logic of the byte-code program as it executes.

There are several different ways in which such an interpreter can implement conditional statements in the input program, which all amount to setting the VIP to one of two alternatives depending on the value of some predicate. VMProtect [41] uses arithmetic on the condition code flags to determine the address of the appropriate VIP value. Since the flags are in general symbolic, this causes the interpreter's VIP to become symbolic as well.

Symbolic execution of virtualized programs becomes challenging if the interpreter's VIP becomes symbolic. The problem is that the constraint solving process used to identify such inputs has no way of distinguishing between alternative execution paths arising due to the interpreter running on a different byte code program, and those arising from a different input to the original byte code program. In effect, symbolic execution turns the interpreter into a generator of inputs it can interpret or accept [5], except that in this case the byte-code is not dependent on the input and so is not itself symbolic—it suffices to make the VIP symbolic. If the VIP becomes symbolic, the number of possible alternatives for the symbolic execution engine to consider at the VM's dispatch point is equal to the number of opcodes in the VM's instruction set. The resulting search space contains all the programs the interpreter is capable of running and exploring it exhaustively is impractical even for small interpreters. Furthermore, even if we hypothesize a successful exploration of the search space, i.e., discovering all of the interpreted programs executable by the interpreter, it is only the interpreter whose execution paths are fully explored, not the interpreted byte-code. Note that this is a more general situation than handling of symbolic memory addresses although the issue with symbolic addresses is still a problem with symbolic execution engines [9,33].

It is not difficult to cause the VIP to become symbolic: all that is needed is to make the VIP input dependent at some point, e.g., by transforming control dependencies into direct data dependencies. Moreover, this attack against symbolic execution can be used with arbitrary predicates, which makes it more flexible than that of Sharif *et al.* [36], which is restricted to equality predicates.

3.2 Conditional Jump to Conditional Jump Transformation

The previous section discussed how an obfuscator could use explicit arithmetic on the condition code flags to turn conditional jumps into indirect jumps that are harder to analyze symbolically. Here we discuss how similar arithmetic operations can be used to transform the predicate associated with a conditional jump to a completely different predicate, as illustrated by the following example.

EXAMPLE 3.2. Consider the following code fragment:

```
1   r0 := input();
2   FLAGS := test(r0)   /* x86:  test */
3   push(FLAGS)         /* x86:  pushf */
4   r1 := pop()
5   r2 := r1 >> 4       /* x86:  shr */
6   push(r2)
7   FLAGS := pop()      /* x86:  popf */
8   jpe L    /* jpe:  jump if parity even */
```

Instructions 2–4 above check the input value and move the condition code flags into register r_1. This register is then right-shifted by four bits (instruction 5) and the resulting value is moved back into the condition code flags (instructions 6, 7), which is used to perform a conditional jump (instruction 8). The conditional branch instruction, `jpe`, is not a very common one: it stands for "*jump if parity is even*" and is taken if the parity flag is set. In reality, however, the bit that is actually being tested is not the parity flag, but rather the bit that was shifted into the parity flag's position by instruction 5—namely, the zero flag. In other words, the the condition that is really being tested is whether the input value read into r_0 by instruction 1 is zero or not; however this is being done using a very different predicate.[2] ∎

The approach illustrated above can also be used to construct opaque predicates, i.e., conditional jumps that are either always taken or always not taken.

The issue described here is orthogonal to that of transforming an input value to a different value and applying a different predicate to the transformed value [36], since it involves using architecture-specific knowledge to transform meta-information. The commercial obfuscation tool EXECryptor [38] uses this approach to produce long sequences of this kind of bit-shuffling operations to hamper analysis.

Note that while concolic analyses have to map conditional jump instructions to predicates on values, reasoning about such bit-level manipulations of condition code flags additionally requires fine-grained taint-tracking. Conventional byte- or word-level taint tracking can lead to significant overtainting in the presence of the sorts of bit manipulation illustrated above. Overtainting occurs when imprecision in taint propagation causes the taint analysis to determine values to be tainted, and deemed to be symbolic, when that are in fact independent of the inputs appear to be dependent on them. Conditional branches on expressions involving such spurious symbolic variables are then treated as candidates for generating inputs that can lead to alternative execution paths, resulting in additional computational load on the constraint solver and degrading the overall performance of the system. In the worst case, a very large number of such spurious symbolic variables and associated conditional branches can use up so much resources that the system crashes or is unable to make progress on identifying inputs that would in fact cause the program to take alternative paths.

3.3 Symbolic Code

Symbolic code can be seen as an extension of a code obfuscation technique commonly used in malware, where the program modifies the code region ahead of the program counter, such that execution then falls into the modified code. Symbolic code extends this idea to carry out the code modification using an input-derived value. The idea is that, if the input meets some appropriate condition, the modified bytes encode a jump instruction to some desired address; otherwise, the modified bytes encode some non-jump instructions. The effect is that execution branches to the target of the jump if and only if the input satisfies that condition. The key characteristic of symbolic code is that this is done without executing an explicit comparison or conditional jump

[2] We use the relatively uncommon and unstealthy `jpe` instruction in this example to highlight how different the predicate of the jump instruction can be from the actual condition on the input value. In practice one would expect the obfuscated code to use more common instructions.

```
    call    get_input()
    cmp     eax, TRIGGER
    jz      L
    call    abort()
L:  call    payload()
```

(*a*) Original code

```
    call    get_input()
    sub     eax, TRIGGER
    add     al, 0xEB
    lea     ebx, L1
    lea     ecx, L2
    sub     ecx, ebx
    mov     ah, cl
    mov     word [L1], ax
    nop
    nop
L1: call    abort()
L2: call    payload()
```

(*b*) Obfuscated code executed with non-trigger input

```
     call     get_input()
     sub      eax, trigger
     add      al, 0xEB
     lea      ebx, L1
     lea      ecx, L2
     sub      ecx, ebx
     mov      ah, cl
     mov      word [L1], ax
pc→  jmp        L2
L1:  call     abort()
L2:  call     payload()
```

(*c*) Obfuscated code executed with trigger input

Figure 3: An example of symbolic code

on an input-derived symbolic value, which means that if the input condition is not satisfied, standard concolic analysis does not see a conditional jump in the instruction stream and therefore does not consider the possibility of an alternate execution path.

Figure 3 shows an example of this approach. Figure 3(a) shows the original code where the behavior of the code is based on an input value. The code in 3(b) shows the obfuscated code statically where the obfuscation tries to hide the control transfer based on some trigger value. The code uses the input value to overwrite an instruction in the code in such a way that the execution results in the control being transferred to a code when the value of the input is the desired one. For other inputs either the instruction constructed is an illegal instruction or the control does not reach the hidden code. 3(c) shows the code where the input triggers the execution of the hidden code. With the input value being the desired value, the computed instruction is a jump which transfers the control to the label `L2`.

Symbolic code is a straightforward variation on an obfuscation technique that has long been used in malware: namely, to modify a few bytes ahead of the execution and have execution fall into the modified bytes. This is illustrated in Figure 4, which shows instructions from the NetSky.aa worm (first encountered in 2004). Figure 4(a) shows the first few instructions from a static disassembly of the code. When this code is executed, the `add` instructions at addresses `0x403e64` and `0x403e68` modify five bytes at address `0x403e6e`; execution then falls into the newly created instructions, thereby installing an exception handler at address `0x5cbc32`, which is then used to field the exception raised via a (deliberate) null-pointer dereference by the `mov` instruction at address `00403e84`. The main difference that the symbolic code technique brings to bear is that the bytes used to create the modified code are input-dependent.

Symbolic code can be used to conceal trigger-based behaviors, i.e., behaviors that are exhibited only under specific external or environmental triggers [3]. Existing proposals for detecting such latent behaviors using symbolic execution assume that the control transfers associated with these triggers rely on conditional branches [3, 13]. Symbolic code can evade such approaches by conditionally creating an unconditional jump instruction, e.g., by using input values to create the modified instruction(s) in such a way that only the desired input (trigger) will result in the desired (malicious) execution, but for the rest of values the malicious part does not get exposed to the analysis. Since the resulting control transfer does not use a conditional branch instruction, existing approaches will not consider it as a candidate for symbolic analysis to identify inputs that can trigger alternative execution paths.

4. HANDLING OBFUSCATIONS

Since the primary focus of this work is to improve concolic analysis of obfuscated code, we do not address other potential problems with concolic analyses, e.g., path selection or dealing with system calls. The key idea behind our approach is to use a combination of bit-level architecture-aware taint analysis, bit-level constraints on symbolic values derived from condition-code flags, and architecture-aware constraint generation, to reason about and identify inputs that can cause different control flow paths to be taken.

4.1 Bit-Level Dynamic Taint Analysis

This section considers dynamic taint analysis, where taint is propagated through the instructions in an execution trace. The same static instruction can give rise to many different instruction instances at runtime, with different operands, results, and condition code flags; dynamic taint analysis treats these different runtime instances differently. To avoid unnecessary repetition, we use the term "instruction" to refer to these dynamic instances of instructions: i.e., different runtime instances of the same static instruction are referred to as different instructions.

Taint propagation algorithms generally propagate taint information at the byte- or word-level, i.e., maintain a taint bit for each byte or word of data. However, this turns out to be too imprecise for our needs: our experiences with obfuscations, e.g., those that use bit manipulations to obfuscate conditional jumps, as discussed in Sections 3.1 and 3.2, indicate that the ability to track taint at the level of individual bits can be crucial for dealing with obfuscated code. We therefore carry out taint propagation at bit level granularity. Additionally, since concolic analysis involves reasoning about the conditions under which different execution paths may be taken, we keep track of taint sources arising from

```
00403e5f    mov eax, 0x403e6e
00403e64    add byte [eax], 0x28
00403e67    inc eax
00403e68    add dword [eax], 0x1234567
00403e6e    nop
00403e6f    retf
00403e70    jbe 0x4c
00403e72    call dword near [eax+0x64]
00403e74    push dword [0x0]
00403e7b    mov [fs:0x0], esp
00403e82    xor eax, eax
00403e84    mov [eax], ecx
```

(a) Static disassembly

```
00403e5f    mov eax, 0x403e6e
00403e64    add byte [eax], 0x28
00403e67    inc eax
00403e68    add dword [eax], 0x1234567
00403e6e    mov eax, 0x5cbc32
00403e73    push eax
00403e74    push dword [fs:0x0]
00403e7b    mov [fs:0x0], esp
00403e82    xor eax, eax
00403e84    mov [eax], ecx
```

(b) Runtime code sequence

Figure 4: Self-modifying code in the NetSky.aa worm

condition code flags. This is done using taint tags or *markings*. Taint markings can be of two kinds:

1. A '*generic taint*' marking that indicates that the taint originated from an input value rather than a condition code flag.

2. A triple $\langle ins, flag, polarity\rangle$ where *ins* refers to (a particular dynamic instance of) an instruction in an execution trace; *flag* encodes a condition code flag; and *polarity* indicates whether the bit that the taint marking refers to has the same value as that of the original flag value it was derived from or whether it has been inverted.

Taint analysis takes as input an execution trace and processes the instructions in order, propagating taint bits and taint markings. For each instruction I, taint is propagated from its inputs to its outputs using a *taint mapping function* that is based on the semantics of I. Values obtained as inputs (e.g., set by system calls) are considered to have all of their bits tainted. For instructions that set condition code flags (which include most arithmetic and logical operations as well as the `test` and `cmp` instructions), if any input operands are tainted then taint is propagated to the flags along with the appropriate taint markings. Let $\ell[i]$ denote the i^{th} bit position of an operand (i.e., location or value) ℓ. Taint propagation for an instruction I in the trace is done as follows:

- If none of the source operands of I are tainted, or if the value of a destination bit $dst[i]$ is fixed and independent of the values of the source operands, then $dst[i]$ is marked 'not tainted'. (In general, it is necessary to take implicit flows into account in order to avoid undertainting [8]. Existing approaches to incorporating implicit information flows into taint analyses [11, 21] can be adapted to our purposes. Since this is not the focus of our work, we do not discuss it further here.)

- Otherwise, if all of the source operands of I have the marking '*generic taint*' then:
  - each non-condition-code destination operand of I gets the taint marking '*generic taint*';
  - each condition code flag f affected by I gets the taint marking $\langle I, f, 1\rangle$.

- Otherwise, for each destination bit $dst[j]$ of I (including condition code flags):
  - if the value of $dst[j]$ can be determined from some particular source operand bit $src[k]$, then:
    - if $dst[j]$ has the same value as $src[k]$ then $dst[j]$ gets the same taint marking as $src[k]$;
    - otherwise $dst[j]$ gets the same taint marking as $src[k]$ but with the polarity reversed.
  - Otherwise: $dst[j]$ is marked *tainted* and its taint mark is determined as follows:
    - Each condition code flag f gets a new tag marking $\langle I, f, 1\rangle$.
    - Each non-condition-code bit gets the mark '*generic taint*'.

We keep track of taint markings in terms of bit values—namely, a condition code flag along with its polarity—to simplify reasoning about code obfuscations that manipulate these bits. However, a taint marking $\langle I, flag, polarity\rangle$ also corresponds to a predicate on one or more values in the computation. Since a particular flag may be set differently by different instruction operations, the specifics of the predicate will depend on the instruction I that set the flag. For example, the `cmp` (compare) and `sub` (subtract) instructions set `CF` if there is a borrow in the result; some forms of the integer multiply instruction `imul` set `CF` if the result of multiplication has been truncated; and some bit-rotate instructions (e.g., `rcl`, `rcr`) include `CF` in the rotation and so set it depending on the bit that is moved into it due to the rotation. Given a taint marking $t \equiv \langle I, f, p\rangle$, we can use the semantics of the instruction I, together with the flag f and the polarity p, to determine the predicate associated with the taint marking t. We refer to this predicate as the *flag condition* for t, written $\mathsf{FlagCond}(t)$.

Taint markings allow us to improve the precision of the taint analysis by identifying operations on bits that originate from the same value. As an example, consider the following instruction sequence:

```
1    r0 := input();
2    FLAGS := test(r0)    /* x86: test */
3    push(FLAGS)          /* x86: pushf */
4    r1 := pop()
5    r2 := !r1            /* x86: neg */
6    r3 := r1 ^ r2        /* x86: xor */
```

In this example, instructions 2–4 check the input value and move the condition code flags into register r_1 (in a real-life example the input might be the result of timing the execution of a fragment of code, and the check might determine whether the value falls within a range indicating that the program is not running within an emulator). Instructions 5–7 then carry out a variety of bit manipulations on the flag bits, e.g., as performed in obfuscation tools such as VMProtect and EXECryptor. In this example, our taint analysis will determine that the bitwise negation operation in instruction 5 flips the bits of r_1 into r_2, which means that, after instruction 5, the low bit of r_2 is different from that of r_1, and therefore that the low bit of r_3 after the xor operation in instruction 6 is necessarily 1. Since the value of the low bit of r_3 is constant and thus independent of the input, it will be marked as untainted.

4.2 Handling Obfuscated Jumps

Given a conditional or indirect jump instruction I that is controlled by a tainted (i.e., symbolic) value, we compute the predicate corresponding to it as follows.

1. Identify the condition code flags that control I:
   - For a conditional jump this is obtained from the jump condition of the instruction.
   - For an unconditional jump this is obtained from the tainted bits in the target address whose taint marking is not '*generic taint*'.

   Denote this set of flags by $C(I)$.

   If $C(I) = \emptyset$ then I is an input-dependent indirect jump that is not dependent any conditional jump in the code. We currently do not handle this case.

2. The predicate corresponding to I is then given by

$$\mathsf{InstrPred}(I) \;=\; \bigwedge_{t \in C(I)} \mathsf{FlagCond}(t)$$

   where $\mathsf{FlagCond}(t)$ is the condition associated with the instruction and condition code flag referred to by t (see the previous section).

Let the path constraint up to the instruction prior to I be π, then the path constraint up to and including I is given by $\pi \wedge \mathsf{InstrPred}(I)$.

Example 4.1. The instruction sequence below is semantically identical to that of Example 3.1. but expressed in x86 syntax to illustrate how the analysis works. A '\$' prefix on an operand, e.g., in instructions 5 and 6, indicates an immediate operand.

```
1  call get_input
2  test eax, eax
3  pushfd
4  pop ebx
5  and ebx, $0x40
6  mov ecx, $0x500000
7  or ebx, ecx
8  jmp ebx
```

Instruction 2 in this sequence assumes the standard calling convention where return values are passed in register `eax`.

The taint propagation goes as follows.

After instruction 1, each bit in `eax` has the taint marking *generic taint*.

After instruction 2, the condition code flags in the `EFLAGS` register are tainted as follows. Bit positions 0, 2, 6, 7, and 11, corresponding to the flags Carry (`CF`), Parity (`PF`), Zero (`ZF`), Sign (`SF`), and Overflow (`OF`), gets the taint markings $\langle 2, \mathtt{CF}, 1\rangle$, $\langle 2, \mathtt{PF}, 1\rangle$, $\langle 2, \mathtt{ZF}, 1\rangle$, $\langle 2, \mathtt{SF}, 1\rangle$, and $\langle 2, \mathtt{OF}, 1\rangle$ respectively (here, the instruction value '2' refers to the position of the instruction that set the flag, and the polarity value 1 indicates that the bit has not been inverted).

The data movement instructions 3 and 4 simply copy the taint marks of their source to their destination. Thus, after instruction 3, the corresponding bits of the top word on the stack get these taint markings, and similarly for the register `ebx` after instruction 4. The resulting taint markings of `ebx` are: $\mathtt{ebx}[0] \mapsto \langle 2, \mathtt{CF}, 1\rangle$; $\mathtt{ebx}[2] \mapsto \langle 2, \mathtt{PF}, 1\rangle$; $\mathtt{ebx}[6] \mapsto \langle 2, \mathtt{ZF}, 1\rangle$; $\mathtt{ebx}[7] \mapsto \langle 2, \mathtt{SF}, 1\rangle$; and $\mathtt{ebx}[11] \mapsto \langle 2, \mathtt{OF}, 1\rangle$.

After instruction 5, the only bit of `ebx` that is tainted is $\mathtt{ebx}[6]$, which has the marking $\mathtt{ebx}[6] \mapsto \langle 2, \mathtt{ZF}, 1\rangle$. After instruction 7, this bit position remains the only tainted bit in `ebx`, with the same taint marking, $\langle 2, \mathtt{ZF}, 1\rangle$. From the semantics of instruction 2, namely, **`test eax, eax`**, the flag condition for this taint marking is that register `eax` is 0. Thus, the instruction predicate for the indirect jump at instruction 8 is that `eax` has the value 0 at instruction 2. $\blacksquare$

In this case, it is possible to reason about the possible values of the tainted bits flowing into the indirect jump, and thereby identify the set of possible targets of the jump. From the perspective of concolic analysis to generate alternative inputs and improve code coverage, this is not really necessary since it is enough to identify the instruction predicate $\mathsf{InstrPred}()$ for the indirect jump. The ability to explicitly identify the other possible targets of such obfuscated jumps can be useful, however, for other related analyses of obfuscated code, such as incremental disassembly [28] and deobfuscation [46].

The following example shows how this approach can deal with obfuscated conditional jumps.

Example 4.2. The code fragment below rephrases Example 3.2 in x86 syntax. A '\$' prefix on an operand, e.g., in instruction 5, indicates an immediate operand.

```
1  call get_input
2  test eax, eax
3  pushfd
4  pop ebx
5  shr ebx, $4
6  push ebx
7  popfd
8  jpe L
```

Instructions 1–4 of this example are the same as in Example 4.1 and their analysis is similar to that shown above. After instruction 4, the taint markings of `ebx` are: $\mathtt{ebx}[0] \mapsto \langle 2, \mathtt{CF}, 1\rangle$; $\mathtt{ebx}[2] \mapsto \langle 2, \mathtt{PF}, 1\rangle$; $\mathtt{ebx}[6] \mapsto \langle 2, \mathtt{ZF}, 1\rangle$; $\mathtt{ebx}[7] \mapsto \langle 2, \mathtt{SF}, 1\rangle$; and $\mathtt{ebx}[11] \mapsto \langle 2, \mathtt{OF}, 1\rangle$.

After instruction 5 (`shr`, shift right), the taint markings for register `ebx` are updated to account for the shift. Thus, $\mathtt{ebx}[2]$ (i.e., bit position 2) gets the taint marking $\langle 2, \mathtt{ZF}, 1\rangle$; $\mathtt{ebx}[3]$ gets $\langle 2, \mathtt{SF}, 1\rangle$; and $\mathtt{ebx}[7]$ gets $\langle 2, \mathtt{OF}, 1\rangle$.

The data movement instructions 6 and 7 then copy the resulting bits from `ebx` to `EFLAGS`, and their taint is propagated correspondingly. In particular, after instruction 7 the condition code flag at EFLAGS[2], namely, PF, gets the taint marking for the corresponding position of ebx, i.e., $\langle 2, \mathtt{ZF}, 1\rangle$.

When the conditional jump in instruction 8 is encountered, the semantics of the jpe instruction specify that it is taken if the PF flag is 1. The taint mark for this flag is $\langle 2, \mathtt{ZF}, 1\rangle$, i.e., (since the polarity on the taint mark is 1) that $\mathtt{ZF} = 1$ from instruction 2. From the semantics of instruction 2, namely, `test eax, eax`, the flag condition for this taint marking is that register eax is 0.

Thus, the instruction predicate for the conditional jump at instruction 8 is that eax has the value 0 at instruction 2. ∎

4.3 Handling Symbolic Code

We detect symbolic code when an instruction writes a tainted value to a memory location that forms part of a subsequently executed instruction I. The way in which such a write is handled depends on which portions of the instruction I become tainted as a result:

- If the opcode byte is tainted, then a different input can cause a different instruction to be written into I's location and subsequently executed. While the total number of other possible opcodes is quite large, for the purposes of reasoning about input-dependent conditional jumps we focus on control transfer instructions. To this end, we construct an instruction predicate that gives, as alternatives for the opcode byte(s) of I, all of the binary opcodes for control transfer instructions (direct and indirect unconditional jumps, conditional jumps, and procedure calls and returns).

  In this case it is also possible for part of all of the operand bytes of I (and possibly the instruction following I) to be overwritten. However, our current imlementation focuses on identifying alternative inputs that can cause a control transfer instruction to be created in place of I since this is fundamental to identifying and exploring alternative execution paths.

- If the opcode byte is not tainted but one or more other bytes of the instruction are overwritten with tainted bits, then the corresponding operands are flagged as tainted as the taint analysis proceeds from that point.

  For example, suppose that an operand is overwritten to become the immediate value 1. If any of the bits involved in this are tainted, then in effect the computation uses the instruction overwriting to conditionally incorporate an input-dependent value into the computation, so the input-dependent value should be consdered tainted.

5. EVALUATIONS

We evaluated the ideas presented in this paper using a prototype system we have implemented. Our system (called ConcoLynx) uses Pin [25] to collect execution traces;[3] these traces are then post-processed to propagate taint from symbolic inputs. We ran our experiments on a Linux machine running Ubuntu operating system with an Intel Core i7 (2.6 GHz) CPU with 8 cores and 6 gigabytes of memory.

We used two sets of programs for our evaluations:

[3]This choice of tracing tool is not fundamental to our approach so any other tracing tool can be adapted by our technique.

- The first set consists of three small programs: *simple-if* (shown in Figure 1(a)); *bin-search*, a binary search program; and *bubble-sort*. The *simple-if* program takes a single input which is marked symbolic in our analysis. We ran *bin-search* on an array of size eight; and only the number to be searched for within this array is marked as symbolic. The *bubble-sort* program was run on an array of size three where all the elements were marked symbolic.

- The second set consists of four malicious programs whose source code we obtained from VX Heavens [42]. Each of these programs demonstrates trigger-based behaviors based on some system calls: *mydoom* and *netsky_ae* both check system time to execute their payload if the current time meets the trigger condition; *assiral* checks whether it is being debugged by calling the API function IsDebuggerPresent() and then takes different execution paths based on the result; and *clibo* checks Windows registry keys to check whether it is running for the first time in the system.

The toy programs in the first set were deliberately chosen to have a small amount of simple but nontrivial control flow, so as to make it easier to separate out the performance and precision effects of code obfuscation on concolic analysis. The small size and simple logical structure of these programs were intended to provide a sort of lower bound on expectations for concolic analyses. The programs in the second set were chosen as representative samples of trigger-based behavior in malware. We used source code because of the requirements of the obfuscation tools listed below. For S2E, the inputs to the programs were annotated with S2E's APIs to introduce symbolic inputs to the programs. For Vine, the desired function calls were hooked to introduce taint to programs. The goal was not so much to examine the latest in trigger-based evasion behaviors in malware, but rather to study the impact of code obfuscation on symbolic execution under carefully controlled experimental conditions.

For each of the programs listed above, we examined the behavior of five executables: the original program together with four obfuscated versions obtained using four commercial obfuscation tools: Code Virtualizer [30], EXECryptor [38], VMProtect [41], and Themida [31]. These obfuscation tools create obfuscated binaries for Windows operating system, so for collecting an execution trace, we ran the obfuscated binaries along with Pin tool on a Windows XP service pack 3 operating system running on VMware workstation. Additionally, we built versions of the first set of test programs to incorporate symbolic code into the program's execution.

We compared ConcoLynx with two symbolic execution systems, S2E [10] and Vine [37]. S2E is based on KLEE and is built on top of the LLVM compiler and can discover program states using symbolic execution and virtualization. Vine is a static analysis tool based on Bitblaze and can analyze traces collected with TEMU [37] where the traces are taint annotated.

We performed two types of analysis. The first experiment looked at the effect of obfuscation on the accuracy and the efficacy of the tools while the second examined the practicality of the tools by looking at the cost that obfuscation imposes on the symbolic analysis in terms of the time of analysis and the number of queries which were submitted to the constraint solver. In our experiments with S2E, we used the concolic execution configuration with two path selection strategies available in S2E: depth-first and random state search.

5.1 Efficacy

System	Program	Obfuscation Tool			
		CV	*EC*	*VM*	*TH*
ConcoLynx	*simple-if*	✓	✓	✓	✓
	bin-search	✓	✓	✓	✓
	bubble-sort	✓	✓	✓	✓
	assiral	✓	✓	✓	–
	clibo	✓	✓	✓	✓
	mydoom	✓	✓	✓	✓
	netsky_ae	✓	✓	✓	✓
Vine	*simple-if*	Err	StpErr	Err	Err
	bin-search	Err	StpErr	Err	Err
	bubble-sort	Err	StpErr	Err	Err
	assiral	Err	StpErr	Err	Err
	clibo	Err	StpErr	Err	Err
	mydoom	Err	StpErr	Err	Err
	netsky_ae	Err	StpErr	Err	Err
S2E (DFS)	*simple-if*	✓	✓	✗	✓
	bin-search	✓	✓	✗	✗
	bubble-sort	✗	✓	✗	✗
	assiral	✓	✓	✗	✗
	clibo	✓	✓	✗	✗
	mydoom	✓	✓	✗	✗
	netsky_ae	✓	✓	✗	✓
S2E (random)	*simple-if*	✓	✓	✗	✗
	bin-search	✗	✓	✗	✗
	bubble-sort	✗	✓	✗	✗
	assiral	✓	✓	✗	✗
	clibo	✓	✓	✗	✗
	mydoom	✓	✓	✗	✗
	netsky_ae	✓	✓	✗	✓

Key: CV: Code Virtualizer; EC: EXECryptor; VM: VMProtect; TH: Themida
✓: tool produced at least one input
✗: timeout or fail to produce any results
StpErr: produced constraints crashed STP
Err: runtime error

Table 1: Efficacy of analysis: Code coverage

5.1.1 Code Coverage

These experiments evaluate the extent to which symbolic execution makes it possible to identify and explore different execution paths in obfuscated programs. Table 1 shows, for each program, whether the symbolic analysis was able to generate any alternative input that would cause the program to take a different execution path than it did in the analysis.

For the programs analyzed with our tool we have manually verified that for each branch point in the program the alternative counter example would lead us to another execution path in the program.

In our experiments, Vine generally failed to produce path constraints on programs obfuscated using Code Virtualizer, VMProtect and Themida: for these programs, it gave error messages and exited and the Errin Table 1 corresponds to this behavior. Vine was able to produce path constraints for programs obfuscated using EXECryptor but even then the constraints created by Vine, crashed the STP that is shipped with the Vine tool.

As can be seen from the table, S2E was not able to produce any test case for programs obfuscated with VMProtect and for most of the programs obfuscated with Themida. Moreover, while S2E was able to generate some test cases for many of the programs protected with Code Virtualizer and EXECryptor, the test cases generated for programs obfuscated using EXECryptor cases were redundant, meaning that S2E generated multiple test cases that all resulted in the same execution path being taken. The generation of such redundant inputs can be a problem because it can use up time and computational resources and thereby slow down the overall progress of analysis of a potentially malicious code sample.

Overall, the results indicate that anti-symbolic obfuscations can significantly hinder multi-path exploration using symbolic analysis.

5.1.2 Symbolic Code

We built versions of our synthetic programs to incorporate symbolic code into their logic. S2E did not detect the symbolic code and so did not generate any inputs that would cause different symbolic code to be generated while our tool was able to detect the symbolic codes and generate appropriate constraints thus generated inputs that would trigger other execution paths in the synthetic programs.

5.2 Cost

Table 2 presents normalized data of the analysis time and the number of path constraint queries submitted to the underlying SMT solver by S2E and ConcoLynx (our tool). Since we were not able to get any useful results out of Vine, it is omitted from the costs table and only the performance data for S2E is given. Our system post-processes an execution trace of the program to generate path constraints, while S2E saves program states whenever it reaches a possible branch point in the program's execution. In order to be able to provide a fair comparison of the systems, the numbers presented here are normalized with respect to unobfuscated programs for each tool. For the *assiral* program obfuscated with Themida, we were not able to execute the binary on our tracing facility: the program crashed while generating the trace so we were unable to apply our tool to this program. We chose 12 hours timeout for our toy programs and 6 hours timeout for malicious codes.

The data in Table 2 lead to the following conclusions:

1. ConcoLynx is able to identify the branch points of the obfuscated programs.

   The total number of queries submitted to the constraint solver by our system is seen to go up for programs obfuscated using EXECryptor: for these programs, we have manually verified that the obfuscation tool inserts additional conditional code that uses tainted values, for example the obfuscation tool inserts additional code that checks the sign of the result of an arithmetic operation where is not checked in the original program, making the symbolic engine produce and send more queries to SMT solver.

Program		*Orig.*	*No. queries (normalized)*				*Analysis time (normalized)*			
			CV	*EC*	*VM*	*TH*	*CV*	*EC*	*VM*	*TH*
ConcoLynx	*simple-if*	1.0	1.0	35	1.0	1.0	5.6	1.9	14.6	112.7
	bin-search	1.0	1.0	13	1.0	1.0	9.7	6.6	19.1	339.3
	bubble-sort	1.0	1.0	14.5	1.0	1.0	3.9	12.9	38.5	152.7
	assiral	1.0	1.0	11	1.0	–	8.8	3.5	63.7	–
	clibo	1.0	1.0	17.6	1.0	1.0	24.8	3.3	123.9	10.8
	mydoom	1.0	1.0	8.6	1.0	1.0	5.7	24.6	63.9	55.9
	netsky_ae	1.0	1.0	8.6	1.0	1.0	7.7	3.6	20.3	27.5
S2E (DFS)	*simple-if*	1.0	19	52.8	2002.1	24.4	4.7	8.5	Timeout	Timeout
	bin-search	1.0	7.9	26.7	1.2	0	Timeout	12.6	2	Timeout
	bubble-sort	1.0	0.1	155.6	0	0	154.6	30.7	Timeout	Timeout
	assiral	1.0	15.1	12.2	123	0	2.1	2.3	8.2	Timeout
	clibo	1.0	10.8	14.7	7	0	2.7	3.5	Timeout	Timeout
	mydoom	1.0	17.2	23.2	0	0	4.1	5	Timeout	Timeout
	netsky_ae	1.0	24.5	22.1	181.5	31	3.6	4.2	Timeout	41.7
S2E (random)	*simple-if*	1.0	19	53.2	3584	49.2	4.2	8.5	847	Timeout
	bin-search	1.0	24.3	26.6	17	40.4	Timeout	14.8	16.5	Timeout
	bubble-sort	1.0	57.7	155.5	62.5	14.8	67.5	74.4	Timeout	Timeout
	assiral	1.0	15.5	12.25	52	239	1.3	1.3	Timeout	Timeout
	clibo	1.0	12.1	14.7	37.2	96.5	3.2	3.9	Timeout	Timeout
	mydoom	1.0	17.2	23.7	35.6	3.9	2.7	3.1	Timeout	Timeout
	netsky_ae	1.0	24.5	22.2	31	74.6	2.5	2.5	Timeout	40.6

Key: CV: Code Virtualizer; EC: EXECryptor; VM: VMProtect; TH: Themida

Table 2: Cost analysis of ConcoLynx compared to S2E for obfuscated programs, numbers are normalized to the cost of unobfuscated programs

The increase in analysis time for the obfuscated code ranges from a low of about 2× for Code Virtualizer to a maximum of about 340× for one Themida-obfuscated program. This increase is due primarily to the larger number of instructions executed by the obfuscated programs.

2. S2E is able to successfully analyze most of the programs obfuscated using Code Virtualizer (except for a timeout on *bin-search*) and EXECryptor (except for a failure on *bin-search* with the Random search strategy). This result is encouraging. For VMProtect and Themida, however, S2E failed or timed out on most of the test programs.

Table 3 gives the actual amount of time of the analysis. In Table 3, T_0 is the time for each program to execute without tracing, T_1 shows the time that is needed to collect an execution trace for each program and T_2 and T_3 show the analysis time of conducting standard byte-level taint analysis and bit-level taint analysis (see Section 4.1) respectively on each execution trace. The last four columns in the table show the overhead ratio for each of the above analyses: T_1/T_0 is the overhead of recording the execution trace of a program compared to the native execution time. This overhead ranges from 1499× to 18203× with a geometric mean of 5540× slowdown. T_2/T_0 and T_3/T_0 show how much overhead different taint analyses algorithms impose compared to the run-time of the program. This overhead for byte-level taint analysis ranges between 0.2× and 21× with geometric mean of 3.2× slowdown and for bit-level taint analysis ranges from 2.6× to 191× with geometric mean of 26.1× slowdown. Finally, T_3/T_2 refers to the overhead of bit-level taint over byte-level taint analysis which ranges between 3× to 15.7× with geometric mean of 8.06× slowdown.

The numbers shown in Table 3 suggest that although the taint approach used in ConcoLynx is more expensive than the standard byte-level taint approach, the overhead of tracing a program is significantly higher than the rest of the analyses and so the increased overhead imposed by our taint analysis is dominated by that of trace recording. Moreover, the run-time of our approach, including the overhead of bit-level taint analysis, is much better than the running-time of other tools (S2E and Vine) when dealing with obfuscated code which makes our approach more practical.

6. RELATED WORK

There is a considerable body of research on symbolic and concolic execution: Schwartz *et al.* [33] give a survey. The analysis of malicious and/or obfuscated code forms a significant application of this technology [1–3,13,27,35,47]. While such techniques can be effective when obfuscation is not an issue (e.g., in environmentally triggered programs where the trigger code uses unobfuscated conditional branches), they fail in the face of obfuscations such as those discussed in this paper. The general problems associated with such analyses, e.g., path explosion or symbolic jumps, are known [33], however most of the research literature do not address them explicitly. This is especially problematic for applications of symbolic and concolic analysis to malware code since these programs are often heavily obfuscated to avoid detection and/or hamper analysis. Furthermore, code obfuscation can raise its own challenges for symbolic and concolic analysis, e.g., the symbolic code problem, discussed earlier, which we have not seen discussed elsewhere in the research literature.

	Program	Time (milliseconds)				Overhead Ratio			
		Native Exec. Time (T_0)	Tracing (T_1)	Tainting		T_1/T_0	T_2/T_0	T_3/T_0	T_3/T_2
				Byte-level (T_2)	Bit-level (T_3)				
CV	*simple-if*	16	66,764	12.15	180.88	4172.50	0.75	11.30	14.88
	binary-search	16	71,573	72.03	465.50	4473.12	4.50	29.09	6.46
	bubble-sort	16	70,686	64.53	996.72	4417.50	4.03	62.29	15.44
	assiral	15	55,764	52.16	466.34	3717.33	3.47	31.08	8.94
	clibo	15	35,804	83.98	855.32	2386.67	5.59	57.02	10.18
	mydoom	16	93,472	23.43	160.42	5841.88	1.46	10.02	6.84
	netsky_ae	15	24,288	41.77	411.36	1618.67	2.78	27.42	9.84
EC	*simple-if*	15	116,187	03.74	35.48	7745.33	0.24	2.36	9.48
	bin-search	16	120,967	76.36	635.38	7560.00	4.77	39.71	8.32
	bubble-sort	16	122,284	202.04	1584.00	7642.50	12.62	99.00	7.84
	assiral	16	104,752	50.31	789.77	6546.88	3.14	49.36	15.69
	clibo	16	88,723	53.78	405.24	5545.00	3.36	25.32	7.53
	mydoom	16	136,400	39.09	451.01	8525.00	2.44	28.18	11.57
	netsky_ae	16	124,735	38.06	306.04	7795.93	2.37	19.12	8.04
TH	*simple-if*	453	679,110	920.47	9632.42	1499.14	2.03	21.26	10.46
	bin-search	469	904,851	956.01	6576.10	1929.32	2.03	14.02	6.87
	bubble-sort	406	728,041	1381.13	11957.30	1793.20	3.40	29.45	8.65
	clibo	125	957,996	187.09	1478.65	7663.92	1.49	11.82	7.90
	mydoom	422	1,000,077	307.47	1820.35	2369.67	0.72	4.31	5.92
	netsky_ae	423	768,828	176.60	1847.81	1817.54	0.41	4.36	10.46
VM	*simple-if*	15	229,510	88.24	728.36	15300.70	5.88	48.55	8.25
	binary-search	16	236,892	207.97	1587.92	14805.60	12.99	99.24	7.63
	bubble-sort	16	236,729	329.33	3056.27	14795.00	20.58	191.01	9.28
	assiral	32	430,615	610.79	1484.02	13456.60	19.08	46.37	2.42
	clibo	47	492,316	587.11	1830.58	10474.70	12.49	38.94	3.11
	mydoom	32	582,506	173.09	935.98	18203.10	5.40	29.24	5.40
	netsky_ae	31	334,940	146.03	975.00	10804.50	4.71	31.45	6.67
Geometric Mean						5540.01	3.22	26.14	8.06

Key: CV: Code Virtualizer; EC: EXECryptor; TH: Themida; VM: VMProtect;

Table 3: Cost of analysis with comparison of byte-level and bit-level taint analysis overheads.

Sharif *et al.* [36] and Wang *et al.* [43] discuss ways to hamper symbolic execution via computations that are difficult to invert.

Concolic analyses typically use taint analysis to distinguish between concrete and symbolic values. More generally, taint analysis finds a variety of uses in security applications [18,19,21,29,45]. General frameworks for dynamic taint analysis include those by Clause *et al* [11] and Schwartz *et al.* [33], but these works do not discuss the specifics of fine-grained bit-level taint analysis. Drewry *et al.* describe a bit-precise taint analysis system named *flayer* [15], but this tool—and taintgrind [44], which is based on it—does not consider using separate taint markings to improve resilience against obfuscations. Cavallarro *et al.* [7] and Sarwar *et al.* [32] discuss techniques to defeat taint analyses.

There is a large body of literature on detection and analysis of obfuscated and malicious code (see, e.g., [12,23,24,40]). None of these works consider ways in which code obfuscations can hamper symbolic analysis.

7. CONCLUSIONS

Although analysis of potentially malicious code is an important application of symbolic and concolic analysis, and malware codes are usually obfuscated to avoid detection and hamper analysis, most of the research on symbolic analysis of malicious code does not consider the impact of code obfuscation on the cost and precision of symbolic and concolic program analysis. This paper investigates the effect of code obfuscation on the efficacy of concolic analysis, focusing on three such obfuscations: two that are known to be used in obfuscation tools that are used by malware, and a third that is a straightforward variation of a transformation used in existing malware. Our experiments suggest that existing concolic analysis techniques are of limited utility against code obfuscations commonly used in malware. We propose a way to mitigate the problem using a combination of fine-grained bit-level taint analysis and architecture-aware constraint generation. Experiments using a prototype implementation indicate that this approach significantly improves the efficacy of symbolic execution on obfuscated code.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) under grants CNS-1115829, CNS-1145913, III-1318343, and CNS-1318955; and the US Department of Defense under grant no. FA2386-14-1-3016.

8. REFERENCES

[1] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering.

In *Proc. 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, Feb. 2009.

[2] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1), Aug. 2006.

[3] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection: Countering the Largest Security Threat*, volume 36, pages 65–88. 2008.

[4] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software*, pages 2–23. Springer, 2005.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[7] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Stony Brook University, Stony Brook, New York*, 2007.

[8] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, July 2008.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2011.

[11] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[12] 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)*, pages 275–284, Oct. 2011.

[13] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 25–36, Oct. 2006.

[14] A. Decker, D. Sancho, M. Goncharov, and R. McArdle. Ilomo: A study of the ilomo/clampi botnet. Technical report, Trend Micro, Aug. 2009.

[15] W. Drewry and T. Ormandy. Flayer: Exposing application internals. *WOOT*, 7:1–9, 2007.

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

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[18] C. Hauser, F. Tronel, L. Mé, and C. J. Fidge. Intrusion detection in distributed systems, an approach based on taint marking. In *Proc. 2013 IEEE International Conference on Communications (ICC)*, pages 1962–1967, 2013.

[19] C. Hauser, F. Tronel, J. F. Reid, and C. J. Fidge. A taint marking approach to confidentiality violation detection. In *Proc. 10th Australasian Information Security Conference (AISC 2012)*, Jan. 30 2012.

[20] Intel. IntelÂő 64 and ia-32 architectures software developer's manual. 2015.

[21] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[24] 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.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

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

[27] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[28] S. Nanda, W. Li, L. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2006.

[29] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

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

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

[32] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for

protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[33] 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*, pages 317–331, 2010.

[34] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, Sept. 2005.

[35] 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.

[36] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. 15th Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[37] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proc. of the 4th International Conference on Information Systems Security*, Dec. 2008.

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

[39] Tora. Devirtualizing FinSpy. `http://linuxch.org/poc2012/Tora,DevirtualizingFinSpy.pdf`.

[40] 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.

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

[42] VX Heavens. Vx heavens, 2011. `http://vx.netlux.org/`.

[43] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pages 210–226. Springer, 2011.

[44] Wei Ming Khoo. Taintgrind: a Valgrind taint analysis tool. `https://github.com/wmkhoo/taintgrind`.

[45] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical report, Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005.

[46] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[47] H. Yin and D. Song. Analysis of trigger conditions and hidden behaviors. In *Automatic Malware Analysis*, SpringerBriefs in Computer Science, pages 59–67. 2013.