Linear Obfuscation to Combat Symbolic Execution

Zhi Wang[1], Jiang Ming[2], Chunfu Jia[1], and Debin Gao[3]

[1] College of Information Technical Science, Nankai University, China
zwang@mail.nankai.edu.cn, cfjia@nankai.edu.cn
[2] College of Information Sciences & Technology, Pennsylvania State University, USA
mingjiangpku@gmail.com
[3] School of Information Systems, Singapore Management University, Singapore
dbgao@smu.edu.sg

Abstract. Trigger-based code (malicious in many cases, but not necessarily) only executes when specific inputs are received. Symbolic execution has been one of the most powerful techniques in discovering such malicious code and analyzing the trigger condition. We propose a novel automatic malware obfuscation technique to make analysis based on symbolic execution difficult. Unlike previously proposed techniques, the obfuscated code from our tool does not use any cryptographic operations and makes use of only linear operations which symbolic execution is believed to be good in analyzing. The obfuscated code incorporates unsolved conjectures and adds a simple loop to the original code, making it less than one hundred bytes longer and hard to be differentiated from normal programs. Evaluation shows that applying symbolic execution to the obfuscated code is inefficient in finding the trigger condition. We discuss strengths and weaknesses of the proposed technique.

Keywords: Software obfuscation, symbolic execution, malware analysis.

1 Introduction

Symbolic execution was proposed as a program analysis technique more than three decades ago [21]. In recent years, symbolic execution has advanced a lot. It is usually combined with dynamic taint analysis and theorem proving, and is becoming a powerful technique in security analysis of software programs. In particular, symbolic execution has been shown to be useful in discovering trigger-based code (malicious in many cases, although not necessarily) and finding the corresponding trigger condition [3].

To fight against the state-of-the-art malware analyzers, Sharif et al. proposed a conditional code obfuscation scheme that obfuscates equality conditions that rely on inputs by introducing one-way hash functions [35]. It was shown that analyzers based on symbolic execution are hard to reason about the value of input that satisfies the equality condition. However, admitted by the authors, using cryptographic functions in the obfuscation might improve malware detection [35].

V. Atluri and C. Diaz (Eds.): ESORICS 2011, LNCS 6879, pp. 210–226, 2011.

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f(a_{i-1}) & \text{for } i > 0 \end{cases}$$

$$\text{where } f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

Fig. 1. Collatz conjecture: a_i will eventually reach 1 regardless of the value of n

In this paper, we challenge the requirement of using cryptographic functions in obfuscation to make symbolic execution difficult, and propose a novel automatic obfuscation technique that makes use of linear unsolved conjectures. There are a few advantages of using only linear operations in the obfuscation without any cryptographic ones. First, the obfuscated code becomes less suspicious in malware detection. The obfuscated code produced by our technique only adds a simple loop to the code, making the resulting obfuscated code similar to legitimate programs that employ, e.g., simple number sorting algorithms. Second, such simple obfuscated code makes it possible for our technique to be combined with other obfuscation and polymorphism techniques to achieve stronger protection. Third, the size of the obfuscated code is less than one hundred bytes longer than the original program.

Many unsolved conjectures (e.g., the Collatz conjecture [23], see Figure 1) involve some simple linear operations on integers that loop for an unknown number of times. Such operations are usually fast and commonly used in basic algorithms in computer science. They are perfect candidates to be used in obfuscations to make symbolic execution difficult because symbolic execution is usually weak and inefficient in analyzing loops, in particular, the number of times the loop body executes [25,19,9,10,8,27,38,30,3].

Another advantage of using these unsolved conjectures is that they can be used to obfuscate inequality conditions, a case the previous proposal is unable to handle [35]. Although some inequality conditions could be transformed to (a set of) equality conditions, it might become impractical when the inequality range is big.

We propose and implement an automatic obfuscator to incorporate unsolved conjectures into trigger conditions in program source code. Extensive evaluations show that symbolic execution would take hundreds of hours in order to figure out the trigger condition.

The rest of the paper is organized as follows. Section 2 discusses the background of symbolic execution, unsolved conjectures, and related work. We detail our threat model and give an overview of the steps in our obfuscation technique in Section 3. Detailed implementation of our obfuscator is explained in Section 4. We show the evaluation results of the obfuscated code and discuss strengths and weaknesses in Section 5. Finally we conclude in Section 7.

2 Background and Related Work

In this section, we briefly discuss existing work on symbolic execution, its application, and limitations in handling loops. We also discuss related work on obfuscating software programs. At the end of the section, we outline some unsolved conjectures in mathematics which we make use of in our obfuscator.

2.1 Symbolic Execution and Its Applications

Forward symbolic execution has been extensively utilized in various security analysis techniques [33]. Automatic testing leverages forward symbolic execution to achieve high code coverage and automatic input generation [9,10,8,19,27,25,34,18]. Most of these applications automatically generate inputs to trigger well-defined bugs, such as integer overflow, memory errors, null pointer dereference, etc. Recent work shows that forward symbolic execution can be used to generate succinct and accurate input signatures or filters to block exploits [6,14,4,5]. Previous work has also proposed several improvements to enhance white-box exploration on the programs that rely on string operations [39,7] and lift the symbolic constraints from the byte level to the protocol level [6]. Malware analysis leverages forward symbolic execution to capture information flows through binaries [12,28,40,2]. Brumley et al. proposed MineSweeper [3] that utilizes static analysis and symbolic execution to detect trigger conditions in malware and trigger-based behavior.

2.2 Limitation of Symbolic Execution in Unrolling Loops

Most existing forward symbolic execution techniques have limitations in traversing branches in a loop, particularly when symbolic variables are used as the bound. Typically, only a fixed number of times or a fixed amount of time is spent to approximate the analysis [25,19,9,10,8,27,38,30,3]. Several approaches improve this loop unrolling strategy. LESE [32] introduces new symbolic variables to represent the number of times each loop executes and links symbolic loop variables to symbolic inputs with known input grammar. RWset [1] prunes redundant loop paths by tracking all the reads and writes performed by the checked code. We exploit this weakness of symbolic execution in handling loops to propose our novel obfuscator that uses only linear operations. In Section 6, we discuss the resilient of our obfuscator to these advancements in dealing with loops.

2.3 Binary Obfuscation

Different approaches for binary obfuscation have been developed, and the main purpose is to improve resistance to static analysis [26,31,29,35,24]. Collberg et al. performs binary obfuscation by code transformation [11]. Popov et al. obfuscates binary by replacing branch instructions with trap and bogus code [31]. Moser et al. propose opaque constants to evade static analysis. The main difference between our approach and existing work is that our goal is to impede forward

symbolic execution. Sharif et al. presented an advanced work to attack symbolic execution by encrypting code that is conditionally dependent on input [35], which is the closest to our approach. However encrypting the original code introduces data bytes rarely observed. Our work introduces only linear operations and is less susceptible to statistical de-obfuscation techniques.

2.4 Unsolved Conjectures

Unsolved conjectures are unproven propositions or theorems that appear to be correct and have not been disproven. The Collatz conjecture, also known as the $3x + 1$ conjecture [23] asserts that starting from any positive integer n (see Figure 1), repeated iterations of this function eventually produces the value 1. The $3x + 1$ conjecture and its variations are simple to state but hard to be proven [15,23,20]. Conway proved that such $3x + 1$ problems can be formally undecidable [13]. The $3x + 1$ conjecture has been tested and found to always reach 1 for all integers $\leq 20 \cdot 2^{58}$ in 2009 [37].

Some other examples of unsolved conjectures that we can use in our obfuscator include

$5x + 1$ conjecture:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \ (mod\ 2) \\ n/3 & \text{if } n \equiv 0 \ (mod\ 3) \\ 3n+1 & \text{else} \end{cases}$$

$7x + 1$ conjecture:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \ (mod\ 2) \\ n/3 & \text{if } n \equiv 0 \ (mod\ 3) \\ n/5 & \text{if } n \equiv 0 \ (mod\ 5) \\ 7n+1 & \text{else} \end{cases}$$

Matthews conjecture:

$$f(n) = \begin{cases} 7n+3 & \text{if } n \equiv 0 \ (mod\ 3) \\ (7n+2)/3 & \text{if } n \equiv 1 \ (mod\ 3) \\ (n-2)/2 & \text{if } n \equiv 2 \ (mod\ 3) \end{cases}$$

Juggler Sequence:

$$a_i = \begin{cases} \lfloor a_{i-1}^{1/2} \rfloor & \text{if } a_{i-1} \equiv 0 \ (mod\ 2) \\ \lfloor a_{i-1}^{3/2} \rfloor & \text{if } a_{i-1} \equiv 1 \ (mod\ 2) \end{cases}$$

These conjectures are similar to the Collatz conjecture in that they all converge to a fixed value regardless of the value of the starting integer, see Table 1.

Table 1. Convergence of unsolved conjectures

Conjecture	Convergence	Modular	Operand
$3x + 1$	1	mod 2	Integer
$5x + 1$	1	mod 2, 3	Integer
$7x + 1$	1	mod 2, 3, 5	Integer
Matthews	0	mod 3	Integer
Juggler	1	mod 2	Floating point, Integer

3 Overview of Our Obfuscator

Our proposed obfuscation technique complicates symbolic execution by introducing a spurious input variable and a loop from unsolved conjectures. The additional spurious input variable affects the control flow of the program in such a way that the trigger condition of the malicious behavior depends on this newly added input variable. Therefore, the additional input variable has to be modeled

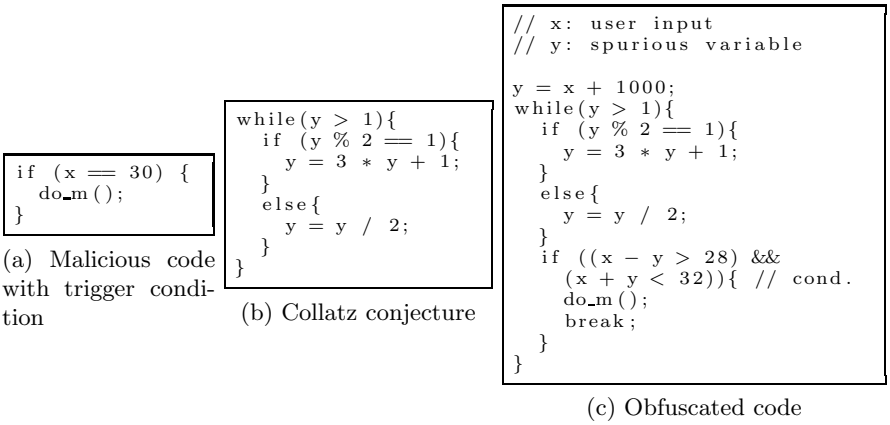
as a symbol in symbolic execution. A loop introduced by an unsolvable conjecture is added to the control flow, typically at the trigger condition. This loop adds a huge number of possible execution paths (growing exponentially) to the program execution, and takes symbolic execution a long time to figure out the original trigger condition.

Note that the introduction of the spurious input variable and the unsolved conjecture do not hide the malicious behavior, a goal some existing obfuscator tries to achieve [35]. What our obfuscator tries to do is to hide the condition under which the malicious behavior is triggered, but not the behavior itself, although our obfuscator could be used in conjunction with other tools to achieve both. In addition, we try to hide the trigger condition without using cryptographic operations so that the obfuscated code is less suspicious.

Figure 2 demonstrates the idea with a simple example. Figure 2a shows a simple code segment where `do_m()` is some malicious behavior with a trigger condition `x==30`. This is a program easily analyzed with symbolic execution. Figure 2b shows a code segment with a while loop. In the loop body, the variable `y` is updated in different ways according to certain condition on `y`. This code segment is hard to be analyzed by symbolic execution because the value of `y` depends on the number of times the loop body gets executed, which is hard to be figured out.

Now if we try to obfuscate the code segment in Figure 2a by introducing a spurious variable and a loop as shown in Figure 2b, we can see that the trigger condition of the malicious behavior is no longer static but depends on the spurious variable, whose value depends on the number of times the body executes (see Figure 2c). Intuitively, this is hard to analyze with symbolic execution.

However, there is one important issue we have not discussed — how do we make sure that the semantics of the program does not change after the obfuscation. In other words, although symbolic execution has a hard time figuring out the number of times the body executes in the loop, are we (as the programmer)



(a) Malicious code with trigger condition

(b) Collatz conjecture

(c) Obfuscated code

Fig. 2. An example

able to figure that out? Answer is yes, thanks to the unsolved conjecture as shown in Figure 1. This unsolved conjecture simply says that `y` will converge to 1 regardless of its initial value. With this, we can work out the condition (`cond.` as in Figure 2c) to be introduced in the obfuscated code under which the malicious behavior executes.

Note that our proposed obfuscator might be susceptible to pattern recognition, assuming that the unsolved conjecture we use is known. This could be solved by randomly choosing various unsolved conjectures, variations to `cond.` according to the particular unsolved conjecture used, or combining with other existing obfuscation techniques (e.g., opaque constants [29]). We discuss this further in Section 6.

4 Implementation

Having explained the basic idea as introducing a spurious input variable and adding a loop from unsolved conjectures, we turn to the implementation details in this section. Here we assume that the source code is available for obfuscation. The same idea can be easily applied to binaries since the obfuscated code we insert is simple and involves linear operations only.

4.1 Adding a Spurious Variable

In most cases, only variables derived from program inputs are taken as symbolic variables in symbolic execution [18,10,2]. We therefore have to make inserted spurious variables dependent upon program inputs.

This is not difficult since the Collatz conjecture hold regardless of the initial value of the variable. For example, if we assume that x represents a program input, our spurious variable y can be made dependent on it by $y = x + c$ where c is a constant, or $y = x + gettimeofday()$, or $y = x + rand()$ where the relation between y and x is more complicated.

However, it is not the case that the more complicated the relationship between y and x is, the longer symbolic execution takes. Symbolic execution does not support some complex operations, e.g., pointer manipulations, floating point operations, etc. When it is obvious that the variable is impossible to reason about symbolically, concrete values will be used to simplify the constraints to continue the symbolic execution. Therefore, we want to make the dependence complicated but not to the extent of being skipped by symbolic execution. We use linear polynomial with normal operations in our experiments (see Section 5). In the example shown in Figure 2, we simply use $y = x + 1000$.

4.2 Choosing an Unsolved Conjecture

Some requirements when choosing an unsolved conjecture include

- **Convergent:** the loop converges.
- **Partially decidable:** although no proof exists, it has been tested that the terminating condition is known under certain range.

- **Machine implementable:** it can be easily implemented in common programming languages.
- **Simple/Linear:** the implementation is simple and involves linear operations only.

All the examples shown in Section 2 satisfy these requirements.

Although the objective of our obfuscation is to confuse symbolic execution but not to combat pattern recognition, program analysts who know our obfuscation technique might create signatures of the unsolved conjectures and the corresponding convergence values to de-obfuscate the program. We discuss two ways of introducing variations to make such pattern recognition difficult. Section 6 also discusses the similarity of an implementation of the unsolved conjecture with the implementation of simple arithmetic algorithms.

In order to use the unsolved conjecture to obfuscate the malicious code, we need to insert a trigger condition within the loop under which the malicious behavior will execute. Intuitively, this trigger condition is related to the converge value of the conjecture (see Table 1), which is a constant regardless of the starting integer value. We can introduce variations to the trigger condition, $y == 1$ in the case of Collatz conjecture, by backtracking a few rounds before the loop actually terminates. For example, $y == 2$ can be used as the trigger condition when we backtrack one round, and $y == 4$ for two rounds.

Another variation we can introduce is for the condition of the loop. This condition is actually unimportant as long as it allows the loop to continue before reaching the trigger condition. Therefore, it can be chosen from a large number of options, including conditions on the converge value (e.g., `while (y > 1)`), conditions on the max stopping time of loops (e.g., `for (i=0; i<1000; i++)`), etc. For Collatz conjecture, the stopping times of positive integers from 1 to 2^{31} are all less than 1000.

4.3 Inserting Trigger-Based Malicious Code into the Unsolved Conjecture

Now we have introduced a new spurious variable $y = x + 1000$ (Section 4.1) and an unsolved conjecture with a trigger condition $y == 1$ (Section 4.2). The next is to insert the malicious code into the unsolved conjecture and to modify the trigger condition accordingly to preserve the semantics of the original code. Depending on the original trigger condition of the malicious code, we modify it in three different ways.

- $>$ **or** $\geq$ **(e.g.,** $x \geq 30$**):** Since the spurious variable is always greater than or equal to 1 in the loop, $x - y \geq 30 - 1$.
- $<$ **or** $\leq$ **(e.g.,** $x \leq 30$**):** Similarly, we have $x + y \leq 30 + 1$.
- $==$ **(e.g.,** $x == 30$**):** This is equivalent to the intersection of two inequalities $x \geq 30 \cap x \leq 30$, and therefore we have $x+y \geq 31 \cap x-y \leq 29$ (also equivalent with the code segment shown in Figure 2c).

We implement the automatic obfuscator which takes input the original source code in C and output obfuscated code to be compiled. We apply the obfuscator on

Table 2. Overhead in size of obfuscated binaries

Malware	Size of original binary	Increase in size (bytes) after obfuscation	
		Before memory alignment	After memory alignment
Blaster	29,426	72	64
Mydoom	28,240	46	64
NetSky	36,182	60	64

three malware samples, Blaster [22], MyDoom [16], and NetSky [36] to evaluate the overhead in size in the obfuscated binaries. Table 2 shows the results.

Blaster is a worm that exploits the DCOM RPC vulnerability. It only triggers its malicious behavior (DoS attack against `windowsupdate.com`) if the system date falls into the range of Aug 16 and Aug 31. The trigger condition in the original code is implemented by two `if` statements which are both obfuscated by our technique. We find that the obfuscated binary code increases by 72 and 64 bytes in its size before and after memory alignment, respectively.

Mydoom is a mass-mailing worm that performs a DoS attack on Feb 1, 2004 starting at 16:09:18 UTC. NetSky is another mass-mailing worm that uses its own SMTP engine to send itself to the email addresses it finds on compromised systems. It only launches the attack on Oct 11 2004. We obfuscate the trigger condition (date and time) for these two malware samples, and find that the increase in size is also 64 bytes after memory alignment, which is hardly noticeable.

5 Security Evaluation

In this section, we first test the effectiveness of our obfuscation on an example with one branch condition, the running example as shown in Figure 2, to evaluate its resistance to symbolic execution. Although this example is very simple, results show that there is very little likelihood to find the trigger input by reasoning about the obfuscated code symbolically. We then continue to discuss if the evaluation results on such a simple example are general in other more complicated programs. We used a machine with a six-core processor running at 3.0 GHz and 4 GB of RAM to do symbolic execution.

5.1 Strategy Used by Program Analyzers

In order to evaluate the effectiveness of our obfuscator in confusing automatic program analyzers that employ symbolic execution, we first discuss the strategy used by the automatic program analyzer.

Recall that the objective of our obfuscation is to hide the trigger condition but not the malicious behavior. Also recall we assume that the program analyzer does not find out the trigger condition by pattern matching due to the variations we introduce to the trigger condition; see Section 4.2. An automatic program analyzer's strategy is described as follows.

1. Pick an initial program input y_0;
2. Dynamically monitor the execution of the program under the chosen input y_i. If the malicious behavior is observed, the trigger condition is found to be y_i;
3. Collect the branch conditions along the execution trace and negate the last condition on the trace;
4. Use a solver to solve for a new program input that satisfies the new sequence of conditions (with the last one negated) as well as the immediate condition of the malicious behavior (`cond.` as in Figure 2c). Let y_{i+1} equal to the new input if it can be found and go to step 2; if the new input cannot be found, negate the next (towards the start of the sequence) condition and send it to the solver until a new input can be found.

In the example as shown in Figure 2c, the trigger condition is `y == 1030` (i.e., `x == 30`). Assume that the program analyzer picks `y = 1158` as the initial input, which will result in a sequence of true/false results in evaluating the condition `y % 2 == 1` in each iteration of the loop. Table 3 shows the value of `y`, the evaluation result of the condition in some of the iterations for the trigger condition `y == 1030` and an initial input of `y == 1158`.

Table 3. Dynamic traces with an initial input of 1030 and 1158

y = 1030			y = 1158			
iteration	y	y % 2 == 1	iteration	y	y % 2 == 1	STP result
1	1030	false	1	1158	false	
2	515	true	2	579	true	
3	1546	false	3	1738	false	
...	...	...	...	...	...	
9	145	true	9	163	true	
10	436	false	10	490	false	
11	218	false	11	245	true	true
...	...	...	...	...	...	...
123	4	false	30	4	false	false
124	2	false	31	2	false	false
125	1	true	32	1	true	false

Table 3 also shows the result of the solver when the program analyzer tries to find the next input. The STP solver keeps returning false, i.e., cannot find a valid input satisfying the given condition sequence, until iteration 11. Once STP returns the next program input, the program analyzer goes back to step 2 and tries again.

This process terminates until the malicious behavior is observed and the trigger condition is found. The reason why the last condition is negated first is because we assume that the program analyzer is able to guess an initial input that is close to the trigger condition. This is a reasonable assumption since the

trigger condition is usually context dependent. Under such an assumption, the program analyzer would like to choose the next y as one that results in a very similar program execution trace as the earlier one, which has a higher probability of getting closer and closer to the actual trigger condition during the experiment.

5.2 Probability of Finding the Correct Trigger Condition

Assuming that the trigger condition is unique, and the solver always manages to find the next input if there exists one that satisfies the given condition sequence (if multiple ones satisfy the condition sequence, a random one will be returned), we notice that the solver finds the next input exactly at iteration i where for all $j \leq i$ the corresponding looping condition $c_j = (y\%2 == 1)$ evaluates to the same result as in the trigger condition. In the example shown in Table 3, this means that for all $j \in [1, 10]$, c_j evaluates to the same value in both `y == 1030` and `y == 1158` while c_{11} evaluates to different values. This can be proven easily because if the solver finds the next input any earlier, it contradicts with our assumption that the trigger condition is unique.

To discuss the probability of finding the correct trigger condition with symbolic execution, we use the following notations in Table 5.2.

Table 4. Notations used

t	the trigger input (1030 in our example)
x	the program input used by the analyzer
$f(x)$	the number of iterations executed before x converges to 1
$g(x)$	largest i s.t. for all $j \leq i$, c_j is the same for t and x
$s(n)$	the number of different x s.t. $g(x) = n$
$z(x)$	the time taken to find the next input x

Table 5 shows the evaluation of some random x. Intuitively, $f(x)$ gives us an idea how long it takes to finish monitoring the execution of the program under input x. $g(x)$ evaluates how close x is with the trigger input t. The difference between $f(x)$ and $g(x)$ indicates the number of times the solver is invoked before it manages to find the next valid input x. An interesting observation here shows that $z(x)$ is not proportional to $f(x) - g(x)$. This is because the time taken for the solver depends on the complexity of the conditions. $z(x)$ is dominated by the last few tests with complex conditions (closer to $g(x)$), and is therefore mainly dependent on the value of $g(x)$.

The last two columns in Table 5 show the value of $s(n)$. Intuitively, the larger $s(n)$ is, the more likely the solver returns the next input x such that $g(x) = n$. There exists some $g(x)$ values that do not correspond to any possible x ($s(g(x)) = 0$), as shown in Figure 3. Since $s(n) = \sum_{i=n+1}^{g(t)} s(i)$, the nonzero $s(n)$ values decrease by half with the increase of n. Therefore,

$$\Pr\left(g(x_{k+1}) = g(x_k) + n\right) = \frac{1}{2^m}$$

Table 5. Statistics for different initial values of x picked

x	$f(x)$	$g(x)$	$z(x)$	$s(g(x))$	$s(g(x)+1)$
1158	32	10	19.14s	33554431	16777215
17414	142	20	878.4s	262144	131072
1049606	153	31	878.4s	4096	2048
134218758	326	43	5178.9s	32	16
2147484678	179	50	1083.6s	2	1
1030	125	125		1	

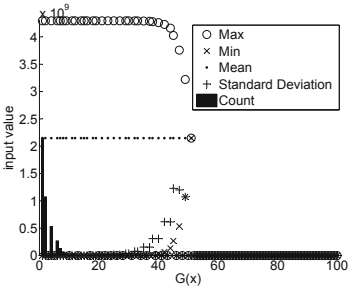


Fig. 3. Distribution of initial inputs for different $g(x)$

where m is the number of nonzero $s(g(x))$ values between $g(x_k)$ and $g(x_{k+1})$.

For example,

$$\Pr(x_{k+1} = 17414 | x_k = 1158) = 262144/33554431 = 0.0078$$

where x_{k+1} is the return of our solver after monitoring the execution with input x_k.

Appendix A shows the continuous scripts of the program analyzer for initial inputs $x = 1158$ and $x = 1034$, which confirms our intuition in the probability of $g(x_{k+1})$ shown above. This shows that it is unlikely the program analyzer gets lucky and the solver returns the trigger condition in the beginning of the study. More likely the program analyzer will take one step closer each time the solver returns an input, just like the scripts shown in Appendix A. The total times needed to find the correct trigger condition are 16871.24 and 21709.1 seconds, respectively, for the two initial inputs.

5.3 Choice of Initial Input

Section 5.2 shows that the program analyzer most likely will take one step closer to the trigger condition t (with an input x that has a slightly larger $g(x)$) every time the solver returns an input for some particular initial input. In this section, we show that the result presented in Section 5.2 can be generalized to other initial inputs. Figure 3 shows the distribution of initial inputs $x \in [1, 2^{32}]$ for different $g(x)$ for the trigger condition $t = 1030$. Appendix B shows that similar distribution is found for eight random values of t.

It is obvious from Figure 3 that there are more initial values with smaller $g(x)$. The number of initial input values continues to drop until it reaches zero for

$g(x) > 50$, with a single exception when $x = t$ which results in $g(x) = g(t) = 125$. Recall that $g(x)$ is an indication of how close x is with t, this means that there are fewer possible values of x when it comes closer to t, which means that the strategy of randomly picking other values of x does not usually give the program analyzer an advantage in finding t. The strategy shown in Section 5.2 is still a reasonably good strategy.

Looking at the mean values, we notice that it is not a continuous line, although the mean is always around 2×10^8. It is not a close line mainly because there exists many $g(x)$ values that do not correspond to any possible x. As we explained earlier, there are many scenarios where the solver might not be able to find any inputs. A closer look into the original data reveals that the mean is always around 2^{16}. This shows that there is very little bias in the distribution of x when t is small, and therefore the program analyzer could not get much advantage by choosing smaller/larger initial values.

This analysis shows that choose different initial inputs does not give the program analyzer significant advantages, and the analysis results in Section 5.2 can be extended to different initial inputs, as well as different trigger conditions (see Appendix B).

6 Limitations

Our obfuscator is designed to make symbolic execution difficult in finding out a trigger condition of malicious code. We show its effectiveness in some examples and its security in Section 5. However, this obfuscator is not designed to solve all obfuscation problems and there are some limitations to it.

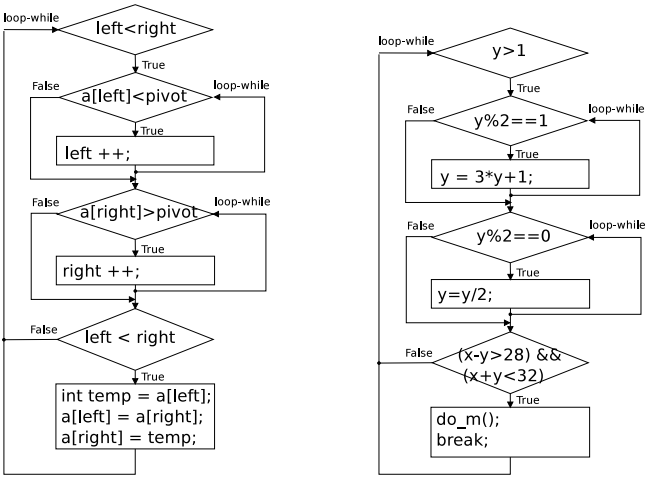
Constants. Our obfuscator is not designed to obfuscate constants. In fact, we introduce additional constants into the obfuscated code. To handle this problem, our obfuscator can be used in conjunction with opaque constants [28] to hide special characteristic of the obfuscation.

Malicious behavior. Our obfuscator is not designed to hide the malicious code, but the condition under which the malicious code will be executed. A malware author can introduce existing code mutation techniques, such as polymorphism and metamorphism, to make it difficult to analyze the malicious behavior.

Pattern matching. The unsolved conjectures introduced by our obfuscator might introduce special patterns that can be identified. Besides using different conjectures shown in the section 2.4 and introducing variations as discussed in Section 4, here we show that the control flow of our unsolved conjectures is very similar to some common program algorithm, which makes pattern matching difficult.

Figure 4 shows that the control flow of the two code segments are similar. We also use one of the most sophisticated binary difference analyzer, BinHunt [17] to analyze the similarity of various binary codes, and show the results in Table 6. Results show that our obfuscated code is very similar to the code of quick sort.

Larger set of triggered inputs. In our analysis we assume that there is a single integer that satisfies the trigger condition, and show that symbolic execution has



(a) A quick sort algorithm (b) Our obfuscated code

Fig. 4. Control flow comparison

Table 6. Binary difference analysis (larger matching strength indicates higher similarity)

Matching strength	Our obfuscated code	Select sort	Bubble sort
Quick sort	0.85	0.49	0.38

a hard time figuring it out. However, the probability results may change when there is a larger set of inputs that satisfy the trigger condition.

Execution overhead. Our obfuscator introduces some additional overhead to the execution of the program due to the loop added. This is usually not a concern when it is applied in a malicious program. However, it may be an issue when the technique is used to obfuscate legitimate programs.

7 Conclusion

In this paper, we introduce a novel obfuscator that makes symbolic execution difficult in finding trigger conditions. Our obfuscator applies the concept of unsolved conjectures and adds a loop to the obfuscated code. Experiments show that symbolic execution will have a hard time unrolling the loop and therefore inefficient in figuring out the trigger condition under which certain code segment will be executed. Our security analysis shows that there does not exist other analyzing strategy in making the analysis simpler, even when different initial inputs are used or when the trigger condition is different.

Acknowledgments. The authors thank the anonymous reviewers for their suggestions. This work was supported by the National Natural Science Foundation

of China under grant 60973141, the Natural Science Foundation of Tianjin under grant 09JCYBJ00300 and the Specialized Research Fund for the Doctoral Program of Higher Education of China under grant 20100031110030.

References

1. Boonstoppel, P., Cadar, C., Engler, D.: RWset: Attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 351–366. Springer, Heidelberg (2008)
2. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin., H.: Bitscope: Automatically dissecting malicious binaries. Technical report cs-07-133, School of Computer Science, Carnegie Mellon University (March 2007)
3. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Analysis in Defense, vol. 36 (2007)
4. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of the 2006 IEEE Symposimu on Security and Privacy (2006)
5. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signature using weakest preconditions. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy (2007)
6. Caballero, J., Liang, Z., Poosankam, P., Song, D.: Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (2009)
7. Caballero, J., McCamant, S., Barth, A., Song, D.: Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Tech. rep., Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley (March 2009)
8. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 2008 USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008 (2008)
9. Cadar, C., Engler, D.: Execution generated test cases: How to make systems code crash itself. In: Proceedings of the 12th SPIN Workshop (2005)
10. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE:automatically generating inputs of death. In: Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS 2006) (2006)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report 148, Department of Computer Sciences, The University of Auckland (1997)
12. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (2010)
13. Conway, J.H.: Unpredictable iterations. In: Proceedings of the 1972 Number Theorey Conference (1972)
14. Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M.: Bouncer: Securing software by blocking bad input. In: Proceedings of the 2007 ACM Symposium on Operating Systems Principles (SOSP) (2007)

15. Crandall, R.E.: On the "3x + 1" problem. Mathematics of Computation 32, 1281–1292 (1978)
16. Ferrie, P.: W32.Mydoom (2004), http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99&tabid=2
17. Gao, D., Reiter, M.K., Song, D.: BinHunt: Automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008)
18. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proceedings of the ACM Conference on Programming Lanuguage Design and Implementation (2005)
19. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (2008)
20. Guy, R.K.: Unsolved problems in number theory. Problem Books in Mathematics (2004)
21. King, J.C.: Symbolic execution and program testing. Commun. ACM 19, 385–394 (1976), http://doi.acm.org/10.1145/360248.360252
22. Knowles, D., Perriott, F.: W32.Blaster (2003), http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99&tabid=2
23. Lagarias, J.C.: The 3x+1 problem and its generations. Amer. Math. Monthly 92, 3–23 (1985)
24. Lee, B., Kim, Y., Kim, J.: binOb+: a framework for potent and stealthy binary obfuscation. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (2010)
25. Lee, G., Morris, J., Parker, K., Bundell, G., Lam, P.: Using symbolic execution to guide test generation. Software Testing, Verification & Reliability 15(1), 41–61 (2005)
26. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (2003)
27. Molnar, D., Li, X., Wagner, D.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the 2009 USENIX Security Symposium (2009)
28. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 2007 USENIX Security Symposium (2007)
29. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proceedings of the 23rd Annual Computer Security Applications Conference (2007)
30. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: Automatic protocol replay by binary analysis. In: Proceedings of the 13th ACM Conference on Computer and and Communications Security (CCS 2006) (2006)
31. Popov, I.V., Debray, S.K., Andrews, G.R.: Binary obfuscation using signals. In: Proceedings of the 2007 USENIX Security Symposium (2007)
32. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (2009)
33. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy (2010)

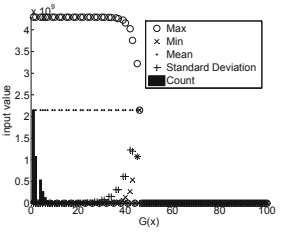
34. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for c. In: Proceedings of 13th International Symposium on the Foundations of Software Engineering, FSE 2005 (2005)
35. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (2008)
36. Shinotsuka, H.: W32.NetSky (2004), http://www.symantec.com/security_response/writeup.jsp?docid=2004-030717-4718-99&tabid=2
37. Silva, T.O.: Computational verification of the 3x+1 conjecture. Tech. rep., Electronics, Telecommunications, and Informatics Department,University of Aveiro (November 2010), http://www.ieeta.pt/~tos/3x+1.html
38. Wang, T., Wei, T., Lin, Z., Zou, W.: Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009) (2009)
39. Xu, R.G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis (2008)
40. Yin, H., Song, D.: Panorama: capturing system-wide information flow for malware detection and analysis. In: ACM Conference on Computer and Communications Security (CCS 2007) (2007)

A Contineous Scripts of the Program Analyzer When $x = 1158$ and $x = 1034$

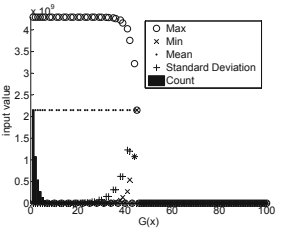
Table 7. Complexity of symbolic formulas and average solving time

Round	x	g(x)	f(x)	z(x)	# of STP nodes	x	g(x)	f(x)	z(x)	# of STP nodes
1	1158	10	32	19.14	19107	1034	3	125	707.6	127543
2	1286	11	27	13.6	17613	1022	5	63	127.6	45602
3	1542	13	35	24.2	24076	550	7	93	464.4	124144
4	2054	14	38	31.2	26042	1222	8	40	38.4	24682
5	3078	15	36	27.3	26119	646	10	101	345.8	65050
6	5126	17	55	72.2	40000	4358	11	47	64.8	29976
7	9222	18	110	469.2	84462	518	13	124	621.6	125150
8	17414	20	142	878.4	110404	7174	15	120	598.5	128339
9	33798	22	60	98.8	44059	25606	18	127	654	131941
10	66566	24	56	70.4	41148	50182	20	66	124.2	57248
11	132102	26	101	345	76491	99334	22	116	413.6	85067
12	263174	28	102	288.6	86298	197638	24	148	892.8	145560
13	525318	29	90	244	68720	4326406	26	181	1286.5	150866
14	1049606	31	153	878.4	118753	1573894	29	76	141	58221
15	2098182	32	105	350.4	76268	6292486	32	228	1999.2	185416
16	4195334	34	137	710.7	105035	12583942	34	105	319.5	81474
17	8389638	36	169	824.6	147493	25166854	36	230	1978.8	188020
18	16778246	37	82	126	71580	100664326	39	139	750	138716
19	33555462	39	171	1188	129648	1946158086	41	135	639.2	137371
20	67109894	41	172	956.3	146783	2013266950	43	183	1162	161645
21	134218758	43	326	5178.9	281345	1879049222	44	306	4034.8	243945
22	268436486	44	174	975	147574	1610613766	46	249	2151.8	199218
23	536871942	46	175	993.3	148022	3221226502	48	177	1109.4	153689
24	1073742854	48	176	1024	147881	2147484678	50	179	1083.6	150026
25	2147484678	50	179	1083.6	150026					
Sum of z(x)				16871.24					21709.1	

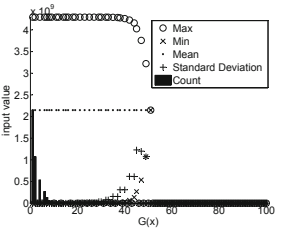
B Distribution of Initial Inputs for Different Trigger Input *t*



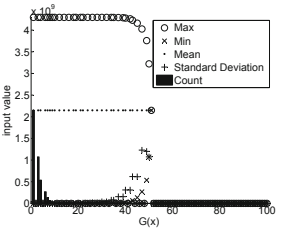
(a) t =0x0000000a



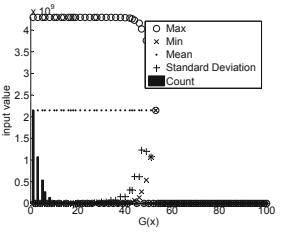
(b) t =0x00000020



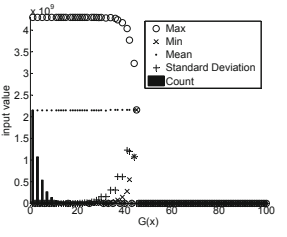
(c) t =0x00000406



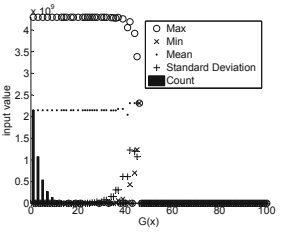
(d) t =0x000057f1



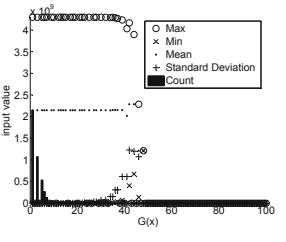
(e) t =0x0007d57b



(f) t =0x00a2355f



(g) t =0x09c45b3f



(h) t =0xc8435f73