Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems		School of Information Systems

9-2014

Control Flow Obfuscation using Neural Network to Fight Concolic Testing

Haoyu Ma

Xinjie Ma

Weijie Liu

Zhipeng Huang

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

See next page for additional authors

DOI: https://doi.org/10.1007/978-3-319-23829-6_21

Citation

Ma, Haoyu; Ma, Xinjie; Liu, Weijie; Huang, Zhipeng; GAO, Debin; and Jia, Chunfu. Control Flow Obfuscation using Neural Network to Fight Concolic Testing. (2014). *International Conference on Security and Privacy in Communication Networks: 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*. 152, 287-304. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/2260

Author
Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin GAO, and Chunfu Jia

Control Flow Obfuscation using Neural Network to Fight Concolic Testing*

Haoyu Ma[1], Xinjie Ma[1], Weijie Liu[1], Zhipeng Huang[1],
Debin Gao[2], and Chunfu Jia[1]

[1] College of Computer and Control Engineering, Nankai University
ma.haoyu@mail.nankai.edu.cn,cfjia@nankai.edu.cn
[2] School of Information Systems, Singapore Management University
dbgao@smu.edu.sg

Abstract. Concolic testing is widely regarded as the state-of-the-art technique in dynamic discovering and analyzing trigger-based behavior in software programs. It uses symbolic execution and an automatic theorem prover to generate new concrete test cases to maximize code coverage for scenarios like software verification and malware analysis. While malicious developers usually try their best to hide malicious executions, there are also circumstances in which legitimate reasons are presented for a program to conceal trigger-based conditions and the corresponding behavior, which leads to the demand of control flow obfuscation techniques. We propose a novel control flow obfuscation design based on the incomprehensibility of artificial neural networks to fight against reverse engineering tools including concolic testing. By training neural networks to simulate conditional behaviors of a program, we manage to precisely replace essential points of a program's control flow with neural network computations. Evaluations show that since the complexity of extracting rules from trained neural networks easily goes beyond the capability of program analysis tools, it is infeasible to apply concolic testing on code obfuscated with our method. Our method also incorporates only basic integer operations and simple loops, thus can be hard to be distinguished from regular programs.

Key words: Software obfuscation, malware analysis, reverse engineering, concolic testing, neural network

1 Introduction

In recent years, advances in reverse engineering techniques have made software verification and malware analysis more and more powerful [1–3]. With the help of dynamic code analysis which is able to trace a program's execution and to monitor branch information along the trail, an analyzer may explore nearly all

* This project is partly supported by the National Key Basic Research Program of China (Grant No.2013CB834204), the National Natural Science Foundation of China (Grant No.61272423) and the Natural Science Foundation of Tianjin (Grant No.14JCYBJC15300).

possible paths for software analysis. A representative technique is *concolic testing* which helps understanding the control flow structure of program routines [4–9]. It performs symbolic execution along a concrete execution path and generates new concrete inputs to maximize code coverage of the tested program, thus could effectively discover trigger-based behavior that leads to malicious execution.

On the other hand, software developers have also realized the fact that environments under which software runs must be assumed to be potentially malicious. For example, in man-at-the-end (MATE) attacks [10], a powerful adversary with full control of the system could thoroughly inspect and analyze the running program. While concolic testing has been proven powerful in security analysis, it also provides a sharp scalpel for attacks like software cracking and piracy.

Many works have been done to provide countermeasures against both static and dynamic code analysis techniques. *Control flow obfuscation*, which aims to confuse the analyzer by complicating programs' control flow structures, has been one of the important approaches [11–16]. Previous works have shown effectiveness against automatic analyzers to a certain extent, yet there are well-documented shortcomings in terms of generality [11, 14, 15] and performance [12–14, 16].

In this paper, we propose a novel control flow obfuscation design that introduces neural networks to execute conditional control transfers. Our method obfuscates a candidate conditional operation by replacing it with a neural network trained to simulate its functionality. The powerful computation capability of neural networks allows our design to work for conditional statements involving all possible algebraic logic. Meanwhile, the well-known complexity in comprehending the rules represented by neural networks [17–19] ensures that the protected behaviors are turned into an unexplainable form, making it next to impossible for theorem provers or constraint solvers to find concrete inputs that lead to the execution paths behind the networks. Hence, our proposed technique disables concolic testing from exploring control flow structure of the protected program.

Our obfuscator applies to programs written in C/C++ and is evaluated with two common concolic testing tools—KLEE [9] and TEMU [20]. The performance of our design is also tested with selected benchmarks from the SPECint-2006 test suite. Results indicate that our method successfully prevents concolic testing from generating test cases to cover the protected conditional paths while introducing only negligible overhead.

The rest of the paper is organized as follows. Section 2 discusses related works on code obfuscation. The main idea as well as some important details are explained in Section 3. The implementation of our obfuscator is given in Section 4. We analyzed the security of our method against possible attacking strategies of adversaries and show the evaluation results in Section 5. Some discussions about the proposed method are given in Section 6. Finally, we give our conclusion in Section 7.

2 Related Works

2.1 Concolic Testing

Concolic testing is a hybrid software verification technique that combines symbolic execution with concrete testing [4], in which program is tested under a concrete execution path, and symbolic execution is used in conjunction with an automated theorem prover (or a constraint solver based on constraint logic programming) to generate new test cases that cover other concrete paths. Concolic testing has been extensively applied in structural exploration and model checking, and received much attention during the past decade [5–9]. Meanwhile, concolic testing also provides a powerful tool for program inspecting, which may lead to further compromise on software integrity.

Nevertheless, several limitations still exist in concolic testing [21], in which we mainly focus on the capability of underlying theorem prover/constraint solver that concolic testing depends highly on. When the constraints of a path go beyond the capability of the solver, concolic testing can no longer perform symbolic execution along it, thus loses the advantage of exploring new behaviors.

2.2 control flow Obfuscation

Control flow obfuscation is one of the major methods of code obfuscation, which aims to make the control flow of a given program difficult to understand [22]. Despite the many efforts made on the subject, existing control flow obfuscation methods either endure a notable tradeoff on performance, or fit only in limited scenarios.

Sharif et al. presented a conditional code obfuscation scheme that uses hash function to protect equal branch conditions [14]. This method is a representative work of using algorithms that is infeasible to be reversely analyzed in protecting program's control logics. Nevertheless, since cryptographic algorithms like hash function are pseudo-random permutations, they are difficult, if not impossible, to be applied in obfuscating branches triggered by input from a continuous interval (e.g., conditions like $>$ or $\leq$), which significantly limits the application of such type of methods. Also, encrypting introduces overhead that cannot be neglected.

Tricks or special mechanisms in programming were also exploited in building control flow obfuscation. There were already attempts to turn control transfers into signals (traps), then introduce dummy transfers and "junk" code to confuse static analysis [15], or to achieve control flow obfuscation via code mobility by computing targets of program's control transfers remotely on a trusted environment at runtime [13]. However, the former cannot be used in protecting conditional logics, while the later looks impractical given that it requires frequent interaction with the remote third-party.

Targeting the drawbacks of specific reverse engineering techniques, such as exploiting the limitation of symbolic execution in solving unrolling loops in [11], seems to be a more promising way. Yet a small regret is that the above design is not built with a provable theoretic basis. In this paper we try to build control flow obfuscation on a proven difficulty that is too hard for theorem provers to solve.

2.3 Neural Network and Rule Extraction

Artificial neural network (ANN) is a connectionist model, consisting of an interconnected group of artificial neurons (as demonstrated in Fig. 1). ANN is known as a highly distributive, fault-tolerant non-linear algorithm with powerful computational capability. During the 1990s, several researches suggested that a feedforward neural network with a single hidden layer (containing finite number of neurons) is a universal function approximator and should be able to simulate arbitrary functions [23, 24]. Meanwhile, in artificial intelligence and machine learning, researchers generally believe that a main weakness of neural network is the absence of a capability to explain either its process to arrive at a specific decision/result, or in general, the knowledge embedded in it in human-comprehensible form [17]. In 1996, Golea [18] studied the intrinsic complexity of the rule-extraction from neural networks and came out with two key results:

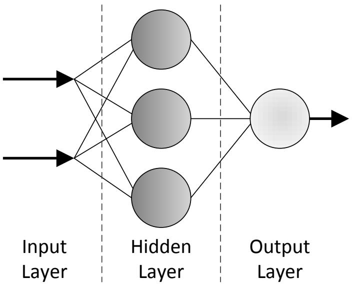


Fig. 1: An example of neural network.

– extracting the minimum Disjunctive Normal Form (DNF) expression from a trained (feed-forward) neural network; and
– extracting both the best monomial rule and the best M-of-N rule [19] from a single perceptron within a trained neural network

are both NP-hard problems. We believe, however, that the incomprehensibility of neural networks, in spite of being treated as a impediment all the time, could actually become an advantage in control flow obfuscation, where the understanding of knowledge in neural networks can be unwanted.

3 Control flow Obfuscation using Neural Networks

In programming, conditional logics are used to selectively transfer control to one of two execution paths, based on whether the value of their inputs satisfy given conditions. Yet from another perspective, as shown in Fig. 2, such selective operations are in some sense equal to a kind of *binomial classification* tasks where:

- all possible values of the input space are assigned to 2 groups — *true*/*false*, each corresponds to a determined execution path;
- the input is examined and classified into one of the groups, then the program is directed to the corresponding path.

This indicates that it is possible to design a control flow obfuscation scheme based on security properties of certain algorithms (e.g. classification) originally used in data mining.

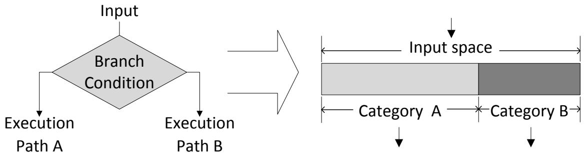


Fig. 2: An intuitive idea of a potential relation between conditional behaviors and the classification task.

We choose neural network as a candidate for building our obfuscation design, not only because it is a well-understood tool in classification, but also due to the incomprehensible nature of its reasoning procedure. The extreme complexity of extracting rules embedded inside neural networks could help providing powerful resistance against reverse engineering techniques that aim to inspect internal structures of obfuscated program routines.

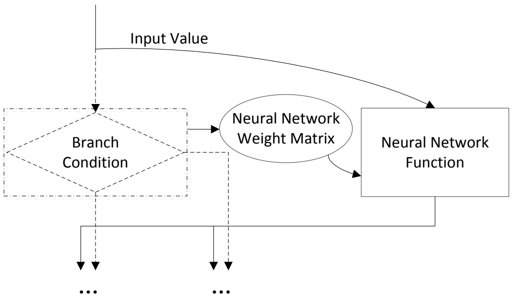


Fig. 3: The framework of control flow obfuscation with neural networks

3.1 Design Overview

The general idea of our method is shown in Fig. 3. The obfuscation takes 2 stages:

At the preparation stage, the obfuscator first locates the target conditional branches in program's source code, and for each of them selects a series of values that trigger both paths to form a training set. It then trains neural networks (in the form of network weight matrixes) which simulate the behavior of the target conditional logics.

After this preparation, the obfuscator goes to the transforming stage, in which it inserts a function to the program to compute output of neural networks, and replaces the target conditional instructions with calls to the neural network function. The embedded function receives the same inputs as the replaced logics, along with the weight matrix of corresponding neural networks, and can then direct execution towards the correct path.

The detailed implementation of our design is a bit more complicated. A few tricks are involved in order to ensure the correctness of obfuscation, as well as enhancing the security.

3.2 Indirect Control Transferring

Similar to some previous control flow obfuscation schemes that transform the subject logics into more complex but semantically equivalent ones [11, 14], the easiest way of replacing a conditional branch with neural network is to attach it with a new conditional logic that instead determines based on whether the network's output is "true". However, considering the capability of neural networks, we can certainly do better than that.

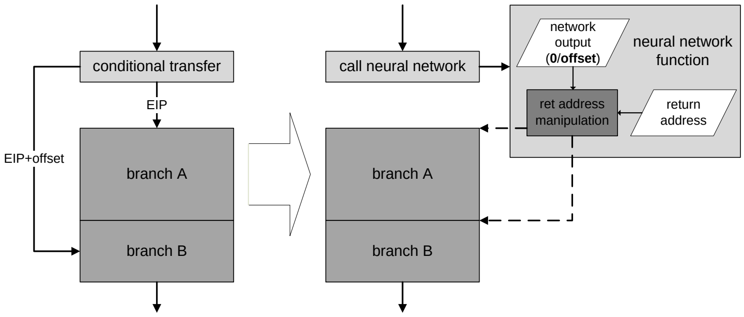


Fig. 4: A demonstration of obfuscating a conditional branch behavior via indirect control transferring manipulated by neural network.

As shown in Fig. 4, a conditional branch is basically to decide whether to jump over a certain code block (or back to a previous one in case of loops) or stay on the code stream, thus the address of target instruction when a branch is taken is usually represented by a relative offset to the value of instruction pointer. Given that neural networks are powerful enough to “remember” any pre-defined output value assigned to each group in classification, it is possible to train the networks to respond with offset of branches, and turn the neural network function into a *conditional dispatcher*. Since replacing a conditional instruction with a call to our dispatcher will automatically push the address of one of the branches into stack for function returning, the dispatcher could control program’s execution path by manipulating its return address according to the output of neural networks. This method could further confuse of analysis tools since it turns conditional logics into indirect control transfers, such semantic level modification could certainly enhance the security of the obfuscation.

3.3 Applying Integer Neural Networks

Due to the unusual *sigmoid* function used as neuron activator and high precision weight values assigned for network connections, traditional neural networks can be quite special and easy to recognize. Although it causes no weakening on the security basis of obfuscation, this does make the embedded networks easy target to be located or traced. However, with integer neural network, we managed to make improvements on this aspect.

```
int stepFunc(int input){
    if(input>=0)
       return 1;
    else
       return -1;
}
```

(a)

```
int stepFunc(int input){

    int x=input>>31;
    x=x*2+1;
    return x;
}
```

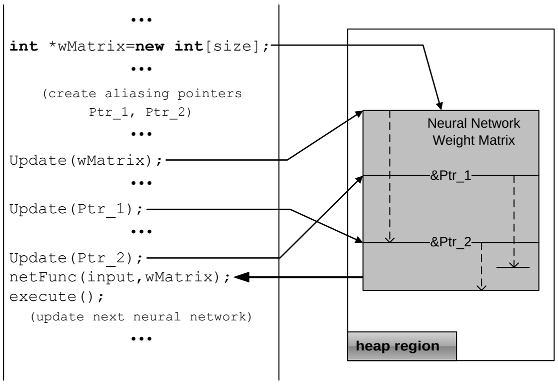
(b)

Fig. 5: The different ways of implementing a step activate function. While (a) is the most intuitive implantation, a equivalently version can be found in (b) where right shifting operation is exploited to turn nonnegative integers into 0 and negative ones into -1, thus gets the same behavior.

Integer neural networks limit their weights to integers only, and apply simple step function (which outputs 1 if the input is equal or greater than 0, or -1 otherwise) as their neuron activator [25, 26]. Although the motivation of the design was simply for getting better performance and enabling the networks to work on devices with limited hardware [27, 28], the fact that integer neural networks consist of only simple operations on common operands gives them an advantage when used in obfuscation, since the simple instruction profile makes them much less significant to potential adversaries. Meanwhile, it is also easier to diversify the actual implementation of methods involve in computing integer neural networks. For example, Fig. 5 demonstrates two different ways of realizing the step function of neural networks. Beside the most intuitive approach that simply returns different values for different inputs, we can also use the side effect of bit-wise shifting to compute the exact same results.

3.4 Dynamic Network Construction

Obfuscating a program with neural networks requires to store the weight matrix of networks for computing. However, if the adversary manages to locate such data, it could directly test the neural networks to reveal what they do, thus avoid analyzing them via reverse engineering techniques. It is impractical to completely prevent such kind of attacks, but approaches for mitigation the problem is certainly possible.



```
             ...
int *wMatrix=new int[size];
             ...
    (create aliasing pointers
          Ptr_1, Ptr_2)
             ...
Update(wMatrix);
             ...
Update(Ptr_1);
             ...
Update(Ptr_2);
netFunc(input,wMatrix);
execute();
   (update next neural network)
             ...
```

Fig. 6: Basic idea of dynamic constructing & updating the neural networks used in obfuscation.

It is known that compared to static data, analyzing heap-allocated objects, especially when pointer aliasing exists, is much difficult [29, 30]. Hence in our design, data of neural networks are created and updated dynamically, as briefly demonstrated in Fig. 6. Doing this has the following benefits that may significantly slow down the adversary's process:

First, when a memory region is allocated for neural networks, pointers can be created targeting different positions in the region to establish complex aliasing effect. Neural network updating can later be carried out using these pointers, creating complicate dynamic data dependencies and making it hard to determine the resulting networks statically.

Second, assuming that all neural networks used in obfuscation share the same topology, they can also share the same memory region. After one network finishes its task, it can be updated into the next one. Each network is only completed right before being queried in control transferring, and are overwritten afterward. Therefore, the neural networks may only be observed correctly when program's execution reaches the corresponding conditional branches.

4 Implementation

We implemented our obfuscator on source code level, using a 3-phase approach. Program is first compiled into binary executable so that static analysis can obtain information of its conditional branches; after that, neural networks are trained for each recorded branch; finally, the obfuscator rewrites program's source code and compiles it into an obfuscated executable.

4.1 Static Analysis

To begin with, the obfuscator must know the exact conditional instructions to be replaced before any transformation actually happens. We do this by compiling the original program sources and statically analyzing the resulting binaries for all conditional jump instructions. Unconditional jumps that are:

- targeting backward at a lower address, or
- followed by the target of a conditional jump/backward unconditional jump

are also recorded since they help complete `if-else`, `while` or `for` structures and thus need to be preserved.

With the help of debug information, these instructions in binaries can be mapped to commands in the corresponding source code. Meanwhile, operands of jump instructions either indicate absolute addresses or relative distances to their targets, both are enough to help determining the offsets for training neural networks. Since code involved in calling a neural network function is longer than that of a conditional branch, the offset for each conditional branch will be adjusted accordingly.

4.2 Neural Network Training

Since integer neural networks sacrifice their precision to some degree due to the data representation, we choose practical swarm optimization (PSO), a sophisticated algorithm that has been widely applied in neural network training [31,32], to ensure the correctness of networks generated by the obfuscator.

As mentioned in Section 3, conditional branches can be replaced with binomial classifications on linear input spaces, which do not cause much trouble to the state-of-the-art training methods. Neural networks' generalization ability also allows them to correctly simulate a function without understanding its complete input-output mapping. Our experience shows that output errors (if any) in the obfuscator only occur on inputs around the branch conditions where values of different groups are close. Therefore, our obfuscator builds the training set of neural networks with all values within the distance of ±1000 to the given branch conditions, along with discretely pick samples from other parts of the input spaces. In case that a branch is triggered by only a few inputs, these values are repeatedly included in the training set to balance the two groups.

Since neural network training starts with a random initial state, the effect of training differs from time to time. Thus after each training, the obfuscator verifies the behavior of the resulting network to see if it matches the conditional logic to be replaced. In case errors are found, the obfuscator goes back to train the network again with a new initial state, until the resulting network passes verification.

4.3 Program Re-writing

After the neural networks are trained, a function that computes their output is inserted into the program, and conditional logics to be obfuscated are replaced by calls to the function. Code for constructing weight matrix of networks are inserted into selected positions of the program according to its control flow graph, to make sure that during execution, all networks will be correctly prepared before being queried. With various arithmetic operations, weight matrix construction can be designed in different ways for each neural network, in order to improve the difficulty in locating them.

5 Evaluation

The major goal of our design is to stop analyses on programs' control flow with automatic tools, and to slow down the adversaries from figuring out the trigger condition of certain code sections, thus we mainly consider 2 attack scenarios against our obfuscation:

1. an adversary could always directly perform concolic testing on an obfuscated program, hoping to reveal certain part of its control flow with the presence of a set of neural networks handling its conditional logics;
2. alternatively, the adversary could try to de-obfuscate the program by first extracting the neural networks via approaches like pattern matching and other static analyses, figure out the conditional logics they represent and remove them to recover the program's original control flow.

We evaluate the effectiveness of our design in both cases, while as another aspect, performances of the obfuscated programs are also tested to show how much overhead is introduced by our method.

5.1 Against Concolic Testing

While it is well-known that concolic testing is limited in solving non-linear algebraic computations, neural networks (even integer neural networks) are typical non-linear algorithms due to the activator used in the neurons. Although neural networks are very different from cryptographic primitives like hash functions (they are not pseudo-random mappings and do not have problems like collision etc.), the difficulty of extracting rules from them [17–19] still ensures that solving a complete set of constraints required to reverse the networks' computation is practically infeasible.

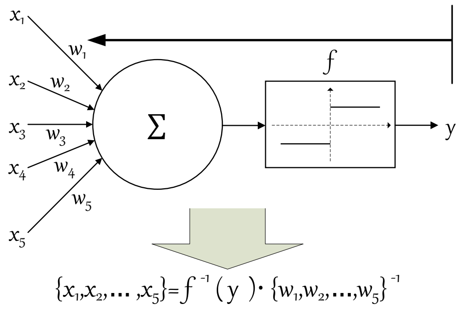

x_1
w_1
x_2
w_2
x_3
w_3
x_4
w_4
w_5
x_5
Σ
f
y
$\{x_1,x_2,\ldots,x_5\}=f^{-1}(y)\cdot\{w_1,w_2,\ldots,w_5\}^{-1}$

Fig. 7: The relation between the structure of artificial neurons in integer neural networks and the algebraic expression of their reverse.

Consider a fully connected feedforward neural network (which is applied in our implementation), during its computing, output of *all* neurons in the current layer of the network (starting form the input layer) are passed to *every* neuron of the next layer as inputs. Therefore, given an output value of a neural network, determining the corresponding input value via automatic analyzer requires to:

1. build the inverse system of the given neural network with all neurons replaced by their reverse formulas and all network flow turned to the opposite direction; and
2. solve this inverse system under the given output value.

However, as shown in Fig. 7, an artificial neuron receives it inputs and compute a weighted sum according to connections defined in the neural network, then transforms it with the neuron activator and gets the output. Thus when reversing a neuron that receives its input from multiple other neurons (which is common in neural networks) at step 1, the analyzer actually gets an underdetermined linear formula of which the number of potential solution is extremely large. Additionally, the analyzer is most likely to encounter chains of underdetermined neurons in neural networks while walking against the direction of network flow, which rapidly amplifies the potential solution space it has to search (growing exponentially). As a result, it is impossible to avoid combination explosion in reversely analyzing neural networks.

To verify our analysis, we performed a simulation to test the effect of our obfuscation against concolic testing on an extremely simple program:

```
void main()
{
    int Var=SomeValue;
    if(Condition(Var))
        Var++;
}
```

We use such a subject program so that the evaluation can be exclusively focused on the effectiveness of applying neural networks in control obfuscation (given that the program does not have other components except the conditional logic to be obfuscated). For the same purpose, we do not apply indirect control transferring to the subject program in this evaluation, to rule out the hindrance in concolic testing caused by indirect branches. For generality, neural networks used in the test are given a series of different topologies, and the branch in the program is given a series of equal/unequal branch conditions (as shown in Table 1).

options of Condition(Var)	network topology		
	inputs	hidden nodes ($\cdots/\cdots$ for multi-layer)	outputs
$> 16, = 16, \leq 29, = 29$	1	10	1
$> 6,\ = 6,\ \leq 11,\ = 11$	1	15	1
$> 4,\ = 4,\ \leq 20,\ = 20$	1	7/8	1
$> 2,\ = 2,\ \leq 13,\ = 13$	1	8/8	1

Table 1: Test case settings for evaluating the effectiveness of our obfuscator.

We selected 2 popular analysis tools for the simulation, respectively K-LEE [9], and TEMU of the Bitbalze platform [20]. KLEE provides powerful path exploration on the source it receives, thus is used to test the effect of our design in impeding analysis that aims to probe unexecuted paths of the program and to determine their trigger conditions. Meanwhile, TEMU works directly on binary executables and performs in-depth concolic testing for execution path verification. It first uses dynamic taint analysis (DTA) to trace an execution path of the program, then generates a constraint set for the traced path and feeds it to the constraint solver, which then solves a test case that is supposed to trigger the given execution path. Although TEMU doesn't actually do path exploration, bringing it into evaluation still provides a convincing demonstration on how our design works against concolic testing.

Our method against KLEE. The analysis from KLEE shows that while the analyzer can easily explore both paths of the original programs and correspondingly generate test cases for them, it can only detect a single feasible path on programs obfuscated by our method. This indicates that our obfuscation successfully hindered concolic testing from understanding the programs' control structures.

Unfortunately we can only go this far since KLEE provides no more information (e.g. errors occurred or unexpected situations happened) to assist the user other than its final analysis result[2]. According to the description in [9], we can only assume that the observed phenomenon is because when KLEE reaches the branch point where the output of the neural network replaces original branch

[2] The output of KLEE includes only the number of paths it discovered along with 1 test case for each path.

condition, it is unable to determine whether both branches are reachable because its constraint solver fails to find a different output from the neural network.

statement (unequal)		> 16	> 6	> 4	> 2	≤ 29	≤ 11	≤ 20	≤ 13
input value		-16							
verification result	original	-16							
	obfuscated	NA							
# of constraints	original	733							
	obfuscated	13751	18759	28931	31584	12809	18759	28931	31658

statement (equal)		$= 16$	$= 6$	$= 4$	$= 2$	$= 29$	$= 11$	$= 20$	$= 13$
input value		16	6	4	2	29	11	20	13
verification result	original	16	6	4	2	29	11	20	13
	obfuscated	NA							
# of constraints	original	2566							
	obfuscated	12809	18759	28931	31584	12809	18759	28931	31584

Table 2: Result of TEMU's execution verification on binary executable of the test cases.

Our method against TEMU. Good thing is that simulations taken on TEMU show positive results in consistence with the assumption we made in the previous section. From Table 2 we can see that for the original programs, TEMU is able to precisely return the input values that cause the execution paths it observes, indicating successful verification. For all obfuscated programs, however, the constraint expressions TEMU generated for the dynamic tainted traces expanded significantly, and it fails to return a valid input value. This result provides more solid evidence indicating that our method managed to make the protected conditional behaviors too complicated for the constraint solvers to reason about.

It should again be emphasized that these evaluations are taken on programs consisting of only the obfuscated control structure. It certainly infers that the complexity of obfuscated control flow would be way beyond existing analysis tools' capability, should our method be applied on actual applications.

5.2 Against Pattern Matching & Brute Force Testing

As mentioned in Section 4.2, since the training process of neural networks starts with an arbitrary initial state, even the neural networks representing exactly the same function may look totally different. To our best knowledge, currently there seems to be no practical method to tell the actual semantic difference between neural networks. Existing rule extraction methods [17, 19] only generate fuzzy and approximate rules to "explain" neural networks, not recovering the exact ones they represent. Consequently, brutally test the input-output behavior of neural networks seems to be a better choice in this attack scenario.

Because our obfuscation still has to be *semantic preserving*, a network's resistance against brute force testing depends highly on the branches being obfuscated. E.g., it is in fact that unequal conditions (e.g. $\leq$ comparisons) being obfuscated could still be revealed by testing the input spaces with simple binary searching, should the corresponding neural networks be successfully located.

However, our obfuscator chooses to use integer neural networks, thus the network computations are implemented with only basic instructions without obvious signatures. The neural network function may also be merged with other operations of the program, in which case it could become even harder to be located.

Furthermore, as described in Section 3.4, our method constructs neural networks dynamically rather than putting them in static data region. It also builds complex pointer aliasing on the network wight matrix region and reuses the same matrix in different networks by updating its values. Therefore, even if the adversary manages to locate the function that computes outputs of neural networks, it is still hard to correctly separate the networks themselves since they are mixed in complicated dynamic data dependencies. The remaining feasible method for the adversary is to monitor the program dynamically and determine one network each time the neural network function is called. It is easy to realize that when enough conditional branches are obfuscated, this forces the adversary to turn to a path-by-path dynamic testing and to solve each unknown neural network he encounters one at a time, thus effectively increasing the complexity of digging the internal structure of the obfuscated program.

5.3 Overhead

The overhead caused by our obfuscation mainly comes from the extra code for updating weight matrixes and computing in neural network function. To evaluate the detail performance of our design, we applied our obfuscator on 5 selected benchmarks from the SPEC-2006 test suite. We obfuscated as many branches as possible in the chosen benchmark programs, in order to get a better picture about the overall performance penalty caused by the obfuscation.

Fig. 8 shows that on all benchmarks being tested, execution overhead caused by obfuscation ranges from around 2% to 20%. The results depend mainly on the number of dynamically taken branches in each execution, rather than the topology of networks[3]. An unfortunate fact is, however, that neural networks (especially when constructed in dynamic way) cause notable memory occupation, as given in Table 3, which might be a small drawback while being applied in practice.

Nevertheless, it is necessary to mention that in this evaluation, we didn't make any kind of optimizations on the code regarding the obfuscation, or use advanced method to select target branches. Practically, obfuscating all possible branches makes the resulting program too suspicious and the neural network

[3] Since the extra execution required by obfuscating each conditional logic is more or less fixed.

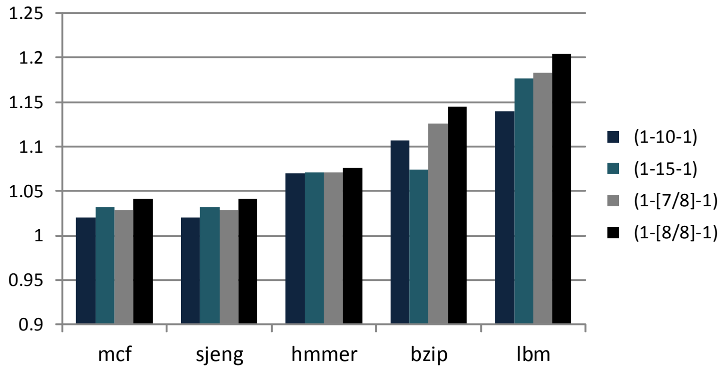


Fig. 8: Normalized execution time of chosen benchmark programs when obfuscated with neural networks of different topologies(against their original version).

computing too obvious. Selectively obfuscating critical points of a program could result in much better performance, with little and acceptable sacrifice on protection strength[4]. Furthermore, data of different neural networks are totally possible to be same partially, thus the updating could be done in a more efficient way. In our future work, we plan to improve our obfuscation system from both aspect, in order to achieve better trade-off between performance and protection strength. But generally, our obfuscator is more than possible to make a program hard enough to be analyzed while causing trivial affect on its performance.

	neural network topology(input-[hidden]-output)			
	1–[10]–1	1–[15]–1	1–[7/8]–1	1–[8/8]–1
memory cost(byte)	622	813	1069	1166

Table 3: Memory cost of different neural network structures required for each obfuscated branch.

6 Discussion

6.1 Scalability on Obfuscating Compound Conditions

In practice it is common to find conditional branches controlled by compound conditions that involve multiple input variables. Intuitively, obfuscating a compound conditional statement needs to compute each of its sub-conditions with

[4] Our experience shows that hiding some conditional behaviors receives much less benefit than doing so on others, e.g. a loop structure is still relatively easier to recognize than a conditional jump, even if obfuscated.

a independent neural network, which can be much more expensive. However it is not hard to understand that algebraic logics of the same type can be equivalently transformed, e.g. $x > A \Leftrightarrow (5 - x) \leq (5 - A)$, or $x = B \Leftrightarrow x + C = D, (D = B + C)$. Therefore, it is possible to "borrow" existing neural networks used elsewhere to participate in obfuscating compound conditional branches.

Consider a branch with compound condition `x==5 && y>0` to be obfuscated. Assume there are already 2 neural networks used for obfuscation in the program: `NetA` representing condition `a!=-1`, and `NetB` representing `b>10`. Also assume that both networks output 0 when respective condition is matched, or the offset of their branch targets otherwise. In such case, we cam simply train a new network `NetC` with condition `sum==0` and the target offset of the compound branch, then replace it by computing `NetC(NetA(x-6)+NetB(y+10))`. This approach allows to keep the memory cost of obfuscating compound branches to the same amount as on simple ones, although we cannot also reduce the corresponding time cost of invoking extra network computing.

6.2 Compatibility with Address Space Randomization

Nowadays' operation systems are in general protected by Address Space Layour Randomization (ASLR) techniques to prevent code injection or other memory error exploiting. ASLR loads executables and public libraries at different random locations for each execution, which affects operations where code pointers are involved. However, as described in Section 3.2, our obfuscator trains its neural networks to remember the relative offset of branch targets, while at the obfuscated branches, calling the neural network function helps to correctly get the base addresses for computing the corresponding branch targets. Since typically ASLR does not disturb the internal structure of program's modules, it will not cause negative impact to our obfuscation.

7 Conclusion

We proposed a novel method based on the complexity of understanding rules embedded in trained neural networks. By training neural networks to simulate selected conditional logics, we managed to direct program's execution path according to the computing of neural networks, while the protected conditional logics can be hidden. Our evaluations demonstrated that applying neural networks in control flow obfuscation significantly increases the difficulty in revealing the obfuscated conditional logics either with concolic testing or using pattern matching and brute force attacks. Simulation on the SPEC benchmarks also indicated that our method is efficient with acceptable memory cost.

We believe that a fresh and interesting view could be opened by this work, indicating that special properties of some well-developed methods in other areas of computer science might be surprisingly useful in designing security applications like control flow obfuscation. Also, our design could be a solid support to the argument that in some cases it is possible to provide strong protection with the absence of tools like cryptographic primitives.

References

1. Lee, G., Morris, J., Parker, K., Bundell, G.A., Lam, P.: Using symbolic execution to guide test generation. Softw. Test. Verif. Rel. **15**(1) (2005) 41–61
2. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the 18th conference on USENIX security symposium (USENIX Security). (2009) 67–82
3. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P). (2007) 231–245
4. Sen, K., Marinov, D., Agha, G.: 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 (ESEC/FSE). (2005) 263–272
5. Sen, K., Agha, G.: Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th International Conference on Computer Aided Verification (CAV). (2006) 419–423
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI). (2005) 213–223
7. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2) (2008) 10:1–10:38
8. Williams, N., Marre, B., Mouy, P.: On-the-fly generation of k-path tests for c functions. In: Proceedings of the 19th IEEE international conference on Automated software engineering (ASE). (2004) 290–293
9. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI). (2008) 209–224
10. Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M.: Guest editors' introduction: Software protection. IEEE Softw. **28**(2) (2011) 24–27
11. Wang, Z., Ming, J., Jia, C., Gao, D.: Linear obfuscation to combat symbolic execution. In: Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS). (2011) 210–226
12. Falcarin, P., Carlo, S.D., Cabutto, A., Garazzino, N., Barberis, D.: Exploiting code mobility for dynamic binary obfuscation. In: Proceedings of the 2011 World Congress on Internet Security (WorldCIS). (2011) 114–120
13. Wang, Z., Jia, C., Liu, M., Yu, X.: Branch obfuscation using code mobility and signal. In: Proceedings of the 36th Annual Computer Software and Applications Conference Workshops (COMPSACW). (2012) 16–20
14. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS). (2008) 321–333

15. Popov, I., Debray, S., Andrews, G.: Binary obfuscation using signals. In: Proceedings of the 16th conference on USENIX security symposium (USENIX Security). (2007) 321–333
16. Schrittwieser, S., Katzenbeisser, S.: Code obfuscation against static and dynamic reverse engineering. In: Proceedings of the 13th International Conference on Information Hiding (IH). (2011) 270–284
17. Tickle, A.B., Andrews, R., Golea, M., Diederich, J.: The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks. IEEE Trans. Neural Netw. **9**(6) (1998) 1057–1068
18. Golea, M.: On the complexity of rule extraction from neural networks and network querying. In: Proceedings of the Rule Extraction From Trained Artificial Neural Networks Workshop, Society For the Study of Artificial Intelligence and Simulation of Behavior Workshop Series (AISB). (1996) 51–59
19. Towell, G.G., Shavlik, J.W.: The extraction of refined rules from knowledge based neural networks. Mach. Learn. **13**(1) (1993) 71–101
20. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security (ICISS). Keynote invited paper. (2008)
21. Qu, X., Robinson, B.: A case study of concolic testing tools and their limitations. In: Proceedings of 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM). (2011) 117–126
22. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscation transformations. Technical Report 148, Department of Computer Science, The University of Auckland (1997)
23. Cybenko, G.: Approximations by superpositions of sigmoidal functions. Math. Control Signals Syst. **2**(4) (1989) 303–314
24. Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural Networks **4**(2) (1991) 251–257
25. Johansson, C., Lansner, A.: Implementing plastic weights in neural networks using low precision arithmetic. Neurocomputing **72**(4–6) (2009) 968–972
26. Tang, C., Kwan, H.K.: Multilayer feedforward neural networks with single powers-of-two weights. IEEE Trans. Signal Processing **41**(8) (1993) 2724–2727
27. Draghici, S.: On the capabilities of neural networks using limited precision weights. Neural Networks **15**(3) (2002) 395–414
28. Moerland, P., Fiesler, E.: Hardware-friendly learning algorithms for neural networks: an overview. In: Proceedings of 5th International Conference on Microelectronics for Neural Networks. (1996) 117–124
29. Ramalingam, G.: The undecidability of aliasing. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(5) (1994) 1467–1471
30. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL). (1996) 1–15
31. Eberhart, R.C., Shi, Y.: Particle swarm optimization: Developments, applications and resources. In: Proceedings of the 2001 Congress on Evolutionary Computation (CEC). (2001) 81–86
32. Zhang, J.R., Zhang, J., Lok, T.M., Lyu, M.R.: A hybrid particle swarm optimization–back-propagation algorithm for feedforward neural network training. Applied Mathematics and Computation **185**(2) (2007) 1026–1037