Multimed Tools Appl (2018) 77:11159–11178
https://doi.org/10.1007/s11042-017-5373-7

Semantic-integrated software watermarking with tamper-proofing

Zhe Chen[1] · **Zhi Wang**[1,2] · **Chunfu Jia**[1,2,3]

Received: 9 May 2017 / Revised: 19 October 2017 / Accepted: 30 October 2017 /
Published online: 10 November 2017

Abstract The existing works of software watermarking have the intrinsic defects: watermarking is independent of program semantics and have weak strength and resilience to state-of-the-art reverse engineering such as symbolic execution, dynamic taint analysis and theorem proving. In this paper, we propose a semantic-integrated watermarking with tamper-proofing to mitigate such problems. This work chooses neural network as the "integrator" and skillfully integrates the watermarking and tamper-proofing module into program semantics. The difficult of reverse engineering or tampering with watermarked program is equal to extracting the rules from neural networks, which had be proven as a NP-hard problem. We have deployed our work in SPECint-2006 benchmarks to evaluate the overhead, strength and resilience. Experiment results show that our watermarking could effectively resist the state-of-the-art reverse engineering, and the introduced overhead is acceptable.

Keywords Dynamic software watermarking · Code obfuscation · Control flow obfuscation · Neural network

This project is partly supported by 973 Program of China(Grant No. 2013CB834204),the National Natural Science Foundation of China (No.61300242, No.61772291), the National Science Foundation of Tianjin (No.15JCQNJC41500, No.17JCZDJC30500), the Open Project Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China(NO. CAAC-ISECCA-201701, NO. CAAC-ISECCA-201702)

✉ Zhi Wang
zwang@nankai.edu.cn

[1] College of Computer and Control Engineering, Nankai University, Tianjin 300350, China

[2] Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China, Tianjin 300300, China

[3] Key Laboratory on High Trusted Information System in Hebei Province, Baoding 071002, China

1 Introduction

It is estimated that 39% of the software in use around the world is illegitimate or unlicensed, with the commercial value of \$10 billion [2]. Such illegitimate or unlicensed software caused serious damage to the financial motivation of the software developers. The root cause of these illegitimate or unlicensed software is Man-At-The-End attacks, known as MATE attacks [1]. MATE attacks have focused on software assets: core code and data. It can be characterized by the following forms: reverse engineering, code reuse, malicious tampering, software piracy et al. The reverse engineering is the foundation of MATE attacks. MATE attacks are more powerful than the Man-In-The-Middle attacks, an attack against traditional cryptography, because MATE attackers are the end users, who could highly control the targeted software. Software publisher cannot distinguish whether their products are running in malicious hosts or normal hosts. As a matter of course, the software intellectual property protection needs to be more effective. This means that the intellectual property owners should be able to claim the ownership of their products. Furthermore, it is possible to prevent the unlicensed users access or destroy the code integrity. *Software watermarking* and *Tamper proofing* can achieve these security goals.

The mainstream categories of software watermarking are *static watermarking* and *dynamic watermarking* [16, 34]. Static watermarking can be embedded and extracted easily; as a double-edged sword, attacker can easily find where the watermark structure stored. So static watermarking has been gradually abandoned. Different from the static watermarking, dynamic watermarking embeds watermark structure into program as forms of data structure or program's execution states. The "dynamic" means that the process of embedding and extracting need running the watermarked programs. According the definition given by Collberg and Clark [16], a complete dynamic software watermarking system has two components: embedder and extractor. There is a program P and a watermark w. The embedder could generate the watermarked program P_w which must preserve the program's origin semantic: for the same input, P and P_w must have the same output. The extractor uses a secret key k to extract the watermark.

$$Embedder(P, w) \rightarrow P_w$$

$$P(x) = P_w(x)$$

$$Extractor(P_w, k) \rightarrow w$$

Assuming that the attacker knows the extracting algorithm, the secret key can prevent attacker from extracting the watermark as easily as intellectual property owner. As a result of the framework about dynamic watermarking, static reverse engineering cannot destroy the dynamic watermarking. However, with the development of reverse engineering, inherent shortcoming of typical dynamic watermarking has become obvious. The defects of existing works are as follows: (1) The semantics of watermark w and program P are mutual independence, which means attacker can remove or distort watermark structure on the premise of ensuring the correctness of programs [46]. (2) The security basis of existing works is the stealth of watermark structure, which lacks the reasonable evaluation of the experiment. (3) Weaken attack tolerance to the state-of-the-art reverse engineering. In order to solve these problems, this paper aims to integrate watermarking into program semantics.

Based on "fragility" is character in both watermarking and software tamper proofing, this work integrates the tamper proofing technique into software watermarking. The tamper-proofing have two modules: checking module and responding module. Checking module collects the execution states of software; these states could be hash of code, execution results or execution environment et al. Responding module will execute a predetermined response if tampering is detected. The possible responses include terminate, restore, degrade results, degrade performance, report attack and punish et al. However, several limitations are still unable to avoid: (1) The responding module always lack stealth, which may lead attacker expense less overhead to locate or remove responding module. (2) The check-sum is mostly stored in explicit mode, which is easy to be modified.

In a similar way, we have mitigated such limitation by integrating tamper-proofing into program semantics.

The primary contribution of this study is presenting a semantic-integrated software watermarking with tamper proofing. Unlike previous studies, our watermarking has been integrated with program semantics, rather than inserting independent code as watermarking. Besides, we have integrated tamper-proofing with program semantics.We choose neural network as the “integrator” to implement watermarking [32]. In summary, Our semantic-integrated watermarking has following advantages:

(1) Different from existing techniques, the security of our watermarking is based on strength of the underlying semantic obfuscation rather than the stealth of watermarking.
(2) Semantic-integrated watermarking is closely integrated with program internal logic, which causes more cost to reverse engineering than existing techniques.
(3) Special implicit tamper responding module that obfuscates judgment and unconditional control transfer has both stealth and resiliency.
(4) Semantic-integrated watermarking has excellent attacking tolerance and acceptable overhead.

The remainder of this paper is organized as follows. We introduce basic tool, semantic obfuscation based on neural network, in Section 2. The design of our watermarking scheme is introduced in Section 3. The implementation detail is in Section 4. We analyze the security of our method against possible attacking strategies in Section 5, and show the evaluation results in Section 6. Section 7 discusses the related works. Finally we discuss the limitation of our scheme and draw a conclusion in Section 8.

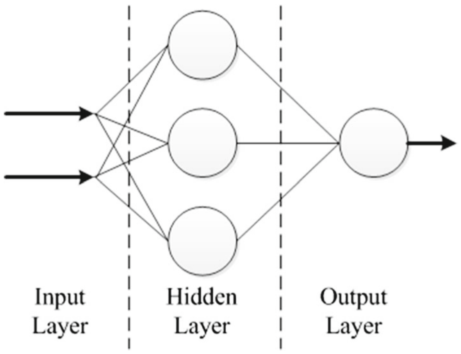
2 Semantic obfuscation based on neural network

In this section, we introduce the basic tool in our study from two aspects. Section 2.1 presents the basic knowledge about neural network. Section 2.2 introduces obfuscating semantics of program branch by using neural network.

2.1 Background of neural network

Artificial neural network (e.g. feedforward neural network,like demonstrated in Fig. 1) is a connection model consisting of an interconnected group of artificial neurons. Its working principle is based on the human thinking process, mainly is used for data classification and linear regression. Neural network is known as a non-linear algorithm with powerful computational capabilities. During the 1990s, several researches had claimed that a neural network of the above representative model is a universal function approximate and should be able to simulate arbitrary functions [24, 25]. In machine learning, a serious weakness of neural networks is that it is difficult to explain networks internal rules in human-comprehensive models; this weakness has partly perplexed researchers. Several researchers had attempted

Fig. 1 An example of neural network



to propose effective algorithm to extract rules from neural network [41, 42]. A representative result gave by Golea [22] is that extracting rules from neural networks is a NP-hard problem. However, as a double-edged sword, the incomprehensibility of neural network could become security foundation in our work.

2.2 Obfuscating the program semantics

In program execution, a conditional branch is used to control program execution path by examining whether the input is conformed to transfer condition. From another perspective to see, the process of branches execution looks like classification in machine learningMa [32]. Figure 2 shows that a condition branch can be considered as a *binomial classification* task, in which all possible values of the input space are classified into two categories, and each category triggers a corresponding path.

According to this feature, neural network can "store" the branch information: inputs set, branch condition and two target addresses. Given enough predefined inputs(branch inputs) and outputs(virtual addresses of corresponding code blocks), a neural network whose semantics is equal to original branch can be trained. Figure 3 shows the framework of neural network obfuscation. The trained neural network replaces the conditional control transfer, and it computes a virtual address according current input. The return module uses the output of neural network to complete the control transfer.

Neural network is chose to build obfuscator, not only because it can provide strong ability of classification, but also its incomprehensibility could help providing resistance

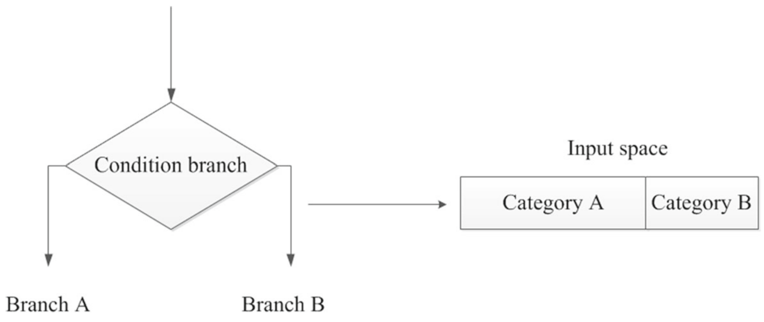


Fig. 2 A condition branch and binomial classification

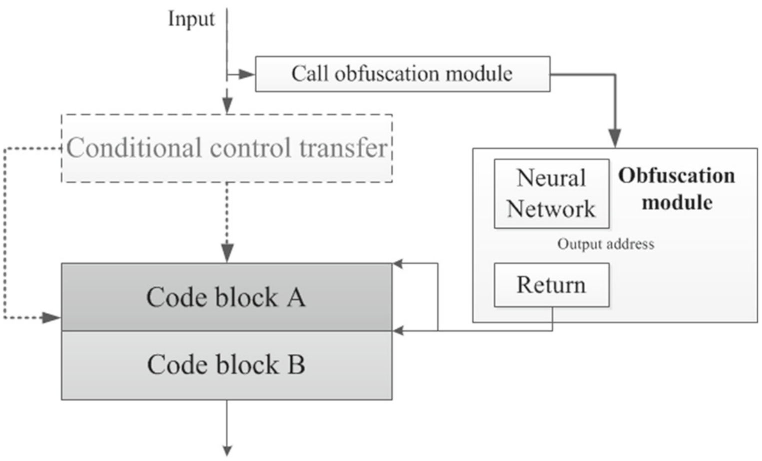


Fig. 3 Semantic obfuscation using neural network

to state-of-art reverse engineering. Because of its incomprehensibility, attacker is difficult to accurately obtain knowledge which is embedded in neural network. Likewise, distinguishing different neural networks is beyond attacker's ability. Ma [32] showed that neural network had provided strong resistance against concolic testing [37], the powerful hybrid software verification technique. The neural network significantly increases the difficulty of reverse engineering. This paper will construct watermarking based on neural network.

3 Design overview

As explained above, existing works about software dynamic watermarking always have the problem that the semantics of watermark and origin program are mutual independent. In fact, typical dynamic watermark is not "embedded" into program; more like "added" to program. The watermarked program feels like $P+w$, not P_w in semantic level. We attempt to find a "integrator" to integrate watermarking and origin program. The proposed scheme utilizes neural network to complicate execution states to make reverse engineering more difficult. The extra states can achieve "express" and "store" watermarking and tamper proofing functionality. This means that our work center on how to use extra states to integrate watermarking into program semantics.

We construct the watermarking framework as a hidden control transfer by deploying semantic obfuscation. As shown in Fig. 4 , we have selected several branches to be obfuscated as watermarked branches. These watermarked branches carry the watermarking fractions in neural network internal rules.

Given a secret key set $\mathcal{K}$ and a watermark $\mathcal{W}$, we first traverse the selected branches in order to confirm watermarking execution trace and branch information. During this process:

- it selects a set of conditional branches $\mathcal{BF} = \{bf_1, bf_2, bf_3..., bf_n\}$ (sequential execution are not necessary in origin program) from program control flow to carry watermark $\mathcal{W}$.
- it collects hash values of corresponding code block $\mathcal{H} = \{h_1, h_2, h_3..., h_n\}$

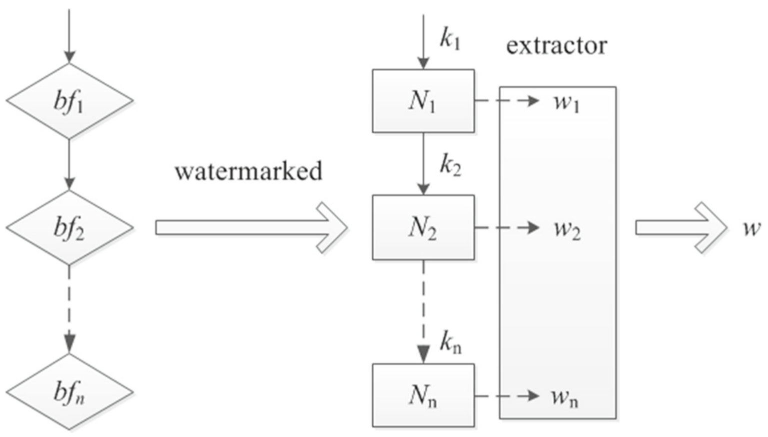


Fig. 4 The framework of watermarking

After the process, watermarking divides $\mathcal{W}$ into n fractions w_1, w_2, w_3..., w_n. Then bf_1, bf_2, bf_3 ..., bf_n can replace corresponding watermarked branches N_1, N_2, N_3 ..., N_n. Each watermarked branch $N_i (i = 1, 2..., n)$ carries a watermark fraction w_i. These watermarked branches not only can complete the equivalent functionality of branches bf_1, bf_2, bf_3 ..., bf_n, but also construct a hidden control transfer which includes watermark fractions. Watermarking extraction can use an external monitor to collect the execution trace by giving software the secret key set $\mathcal{K}$. During this process, watermarking fractions are sequently outputted.

In Fig. 5, it intuitively presents code block layout of the watermarked software in memory. The code blocks of a watermarked branch sequentially locate in memory. The tamper proofing in watermarking has been divide to two modules: checking module and responding module. Checking module is completed by a stealthy algorithm: oblivious hash (OH) [15] to record the hash value of variables. OH can record the change of variables and the execution state of control flow. We insert OH into appropriate location to construct checking module. Later, watermarking provide the implicit response in watermarked branches: the judgment `if(HashValue! =CorrectValue)` is a condition branch and it also can be protected by neural network. Obviously, this response has been designed in that form: it works at the same time as executing the watermarked branch to examine whether tampering has taken place in executed code. The key of stealth response is that neural network only give the response address so that correct hash value is invisible.

In the next three subsections, we present details of the watermarking embedding and extraction processes.

3.1 Extract branch information

As mentioned above, watermarking and tamper proofing need be encoded into extra states, so branches have to be extended for extra execution path as extra states. We define a conditional branch as a 3-tuple: $bf_n = \{\mathcal{I}_n, path_{n-a}, path_{n-b}\}$ in programming; $\mathcal{I}_n$ is the input set and two target addresses: we label them as $path_{n-a}$ and $path_{n-b}$. We extend two extra paths: $path_{n-w}$ and $path_r$. The $path_{n-w}$ point to next watermarked branch bf_{n+1}, and

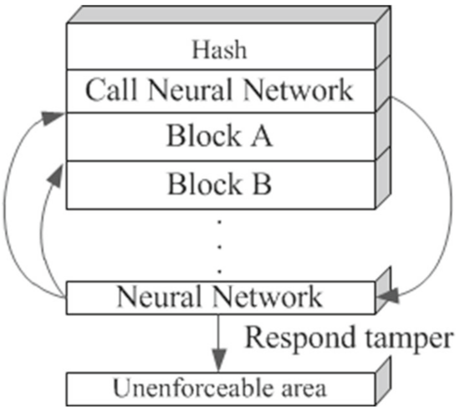


Fig. 5 The location of code block

$path_{n-w}$ is triggered by corresponding secret key k_n. In the same way, all watermarked branches have been extended this type of paths. The $path_{n-w}$ connects two watermarked branches bf_n and bf_{n+1} in order to extract watermark. As response of tampering, $path_r$ always points to the unexecute memory area to terminate the execution when tampering is detected. We integrate value of OH and original input into a two-dimensional vector as new input of watermarked branch. After being watermarked, a watermarked branch has been changed to $bf_n = \{\langle \mathcal{I}_n, h_n \rangle, path_{n-a}, path_{n-b}, path_{n-w}, path_r\}$

3.2 Embed watermarking fraction and tamper proofing module

Considering a 32-bit x86 architecture, the target address of jump instruction always have 24 bits long for long jump (even 8 bits for short jump). This means the output of neural network have 8 bits (24 bits for short jump) are redundancy. These redundancy is the key solution of embedding the watermark. To ensure watermarking can be against code transforming attack well, we take the long jump as control transfer. Although this design may reduce the date-rate of watermarking, it can prove more powerful resiliency than using short jump. We divide a neural network output into four parts for every 8 bits: **Byte0**, **Byte1**, **Byte2** and **Byte3**. As the Fig. 6 shows, we have given the four bytes of different meaning. **Byte1** to **Byte3** is called *transfer section* that store the virtual address for control transfer. **Byte0** is the *watermark section* that stores the watermark fractions. The payload of one watermarked branch is 8 bits. We encode the watermarking fractions into the outputs which have the control transfer point to the $path_{n-w}$; this constructs the extracting path for watermarking recognition.

The training process is the last step for embedding, in which watermarking and tamper proofing are integrated with program's semantics. For concise example, the watermarked branch $bf_n = \{\langle \mathcal{I}_n, h_n \rangle, path_{n-a}, path_{n-b}, path_{n-w}, path_r\}$ will be trained to a neural network. In Table 1, we construct a training set for a watermarked branch to train neural network. The watermark fraction is encoded into $path_{n-w}$.

This table shows that the trained neural network *N* has four categories of output which are used to complete control transfer, store the watermark fraction and respond tampering. The *N* replaces the branch *bf*. In a similar way, all the selected branches can be watermarked.

Watermarking deploys a implicit control transfer to complete the function of branch. The implicit means the target address is implicit in control flow analysis rather than attacker does not disassemble the `JMP` or `CALL` instruction. We choose unconditional jump instruction

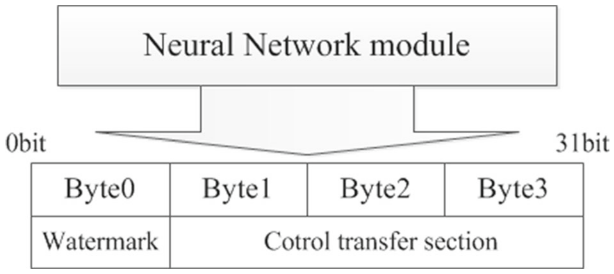


Fig. 6 The way of watermark embed

`jmp eax` to complete control transfer. The return value in EAX only can be observed in run time. Detailed experiment analysis will be presented in Sections 6.1 and 6.2.

3.3 Watermarking extraction

The recognition of watermarking is a dynamic process. Watermarking gives the input set $\mathcal{K}$ and watermarked program P_w to the dynamic monitor. When program P_w runs, dynamic monitor will record the execution trace. The watermarked branches execute sequentially until the last watermarked branch. Collecting 0-7bit of the addresses of $path_{n-w}$, whole watermark is recognized by combine these fractions.

4 Implementation

In this section, we elaborate the implementation details. The watermarking applies to the program written by C/C++ and compiled by common compiler. Meanwhile, a static disassemble is also needed. The implementation begins with calculating the hash value of the protect object. We then insert some place holders to selected branches and use static disassembling to gain the information of extra states. After that, we train the neural networks and replace the place holders by corresponding one.

4.1 Deploy oblivious hash

We deploy OH into software to collect the information of executed trace. In order to balance the level of protection against amount of overhead, we only insert OH to protect the inputs of watermarked branches, which can check the correctness of input variable. We choose MD5 algorithm in OpenSSL as hash function implementation. A global variable is defined to store and update the current hash value.

Table 1 A candidate training samples

Output	Input1	Input2
$path_{n-a}$	$\mathcal{I}_{n-a}$	h_n
$path_{n-b}$	$\mathcal{I}_{n-b}$	h_n
$path_{n-w}$	k_n	h_n
$path_r$	$\mathcal{I}$	h_n

4.2 Neural network training and source codes rewriting

Without loss of generality, let us consider the following code as the a candidate `if-else` condition branches as example:

```
int x;
scanf("%d",&x);
Update(hash, MD5(x));
__asm nop
__asm nop
__asm nop
__asm nop
__asm nop
if(x>5)
    function_a();
else
    function_b();
```

We insert some `nop` as pretreatment placeholder to ensure the relative location between code blocks after being watermarked. After compiling this source code, we disassemble the executable file. The assembly code is like follows:

```
  40100d:   ff 15 a4 20 40 00       call    *0x4020a4
  401013:   83 c4 08                add     $0x8,%esp
  401016:   90                      nop
  401017:   90                      nop
  401018:   90                      nop
  401019:   90                      nop
  40101a:   90                      nop
  40101b:   83 7d fc 05             cmpl    $0x5,-0x4(%ebp)
  40101f:   7e 14                   jle     0x401035
  401021:   68 f4 20 40 00          push    $0x4020f4
  401026:   ff 15 9c 20 40 00       call    *0x40209c
  40102c:   83 c4 04                add     $0x4,%esp
  40102f:   33 c0                   xor     %eax,%eax
  401031:   8b e5                   mov     %ebp,%esp
  401033:   5d                      pop     %ebp
  401034:   c3                      ret
  401035:   68 f8 20 40 00          push    $0x4020f8
  40103a:   ff 15 9c 20 40 00       call    *0x40209c
  401040:   83 c4 04                add     $0x4,%esp
  401043:   33 c0                   xor     %eax,%eax
  401045:   8b e5                   mov     %ebp,%esp
  401047:   5d                      pop     %ebp
  401048:   c3                      ret
```

According to the disassemble code, some needful information can be extracted to train the neural networks. We got the two virtual addresses `0x00401021` and `0x00401035`. They correspond two outputs $path_{1-a}$ and $path_{1-b}$ in the sample training set. For example, we want to embed the letter "*w*" as watermark into the sample program. The ASCII of *w* is 57 in hexadecimal. Assuming the $path_{1-w}$ is `0x00401049`, we encode 57 into **Byte0** of `0x00401049`. The new $path_{1-w}$ is `0x57401049`. When the hash value h_1 has been calculated, we start training process.

We apply practical swarm optimization (PSO), a sophisticated algorithm widely applied in neural network training, as the network learning algorithm [39]. PSO is not only a effective algorithm, but also has better accuracy even in the boundary regions. In fact, no program input is outside of the train set. Consequently, so long as the selected training set is sufficient coverage of boundary values, program can never execute in wrong way. Although the trained model is overfitting, we still welcome for this.

After training process, a neural network model is generated. We package the model into a function `Predict`. This function uses a current input and hash value as its input and output a 32 bits integer. The operation mod is used to get **Byte1** to **Byte3** in outputs. Then we remove the placeholder and watermark the branch. After being watermarked, source code is rewritten like this:

```
int x;
scanf("%d",&x);
Update(hash, MD5(x));
Predict(x, hash) % 0x1000000;
_asm jmp eax
    function_a();
goto label;
    function_b();
label:
```

4.3 Watermarking extraction

In the recognizing process, we select a dynamic simulator: TEMU component in Bitblaze [40] as recognizer. TEMU can provide a dynamic analysis environment through whole-system emulation and dynamic binary instrumentation. When recognizing watermark, TEMU records the execute trace by running the program P_w with the input set $\boldsymbol{K}$. The we analyze the outputs that given by each neural network. If the outputs contain the watermark pieces, the watermark will be successfully extracted.

5 Security analysis

In this section, we have considered the four attack scenarios to watermarking: *brute-force* attack, *additive attack*, *subtractive attack* and *distortive attack*. The tampering attack also has been considered.

Attack model

– Attacker knows the extraction algorithm, but has no information about the secret key $\boldsymbol{K}$.
– Attacker has the state-of-the-art reverse engineering tools, and use them to analyze watermarked program.
– Attacker acquires all privileges to modify the watermarked program.

If attacker can bypass the tamper proofing to remove or distort watermarking, in the meantime, preserve the vast major functionality of program, we say attacker has succeeded.

Brute-force attack In a brute-force attack, attacker simply generates all possible secret keys and try to find the correct one by applying public recognizer algorithm. When attacker knows keys' distribution of possible, he may select more likely ones to test. In our watermarking, we extend original input of watermarked branch to a two-dimensional vector. Only testing all possible inputs not only need amount of computation, program but also terminates if attackers test the wrong hash value. Clearly, the difficulty of brute-force attack is reduced to reverse the hash value. Brute-force attack needs considerable computational overhead so that this attack is impractical.

Additive attack In an *additive attack*, attacker attempts to add his own watermarking w' into watermarked program P_w to generate $P_(w, w')$. If a program has two watermarks, intellectual property owner cannot prove that his watermark is embedded before attacker watermark. In our work, the offset between code blocks needs be strictly fixed in order to ensure program correctness. When attacker has generated $P_(w, w')$, the virtual addresses of code blocks have been changed. $P_(w, w')$ will crash unless attacker can recover original location. The difficulty of recovering is equal to reverse the neural network. Note that, we reject short jump using in watermarked branch because deploying short jump cannot against additive attack in most cases. Short jump uses offset to complete control transfer and its narrow addressing range is only from -128 to $+127$. When attacker inserts his own watermarking, it most possibly maintains the offset of code blocks, which does not affect the correctness of program. Deploying long jump make the larger range to control transfer, which has protected more code blocks to against additive attack.

Subtractive attack In a *subtractive attack*, attacker can (approximatively) point out where the watermark w is hidden and can try to crop it out. After being attacked, intellectual property owner can extract nothing from P_w. This kind of attack must preserve the semantics (at least, preserve needed function). *Subtractive attack* is ineffective because all the watermark fractions have been computed by neural networks; when attacker removes these networks, program cannot run correctly on account of incompleteness. Hence, it is impossible for attacker to remove the watermarking.

Distortive attack In a *distortive attack*, attacker applies semantic-preserving transform (i.e. obfuscation, code compression and encrypt et al.) to scramble program and watermarking. The goal of *distortive attack* is to prevent intellectual property owner to extract watermark. This attack is no need to analyze the location of the watermark, because watermark can be always affected wherever the watermark is hidden. This kind of attack does not work as the same reason of *additive attack* scenario.

Tampering attack Assuming attacker attempts to modify the data and control flow to achieve malicious goal, watermarking always protect this sensitive information. Oblivious hash currently updates the hash value of input variable to ensure the correctness. Meanwhile, implicit response limits the virtual address of code block. On the promise of ignoring analyzing watermarked branches, attacker barely carries his intention because he can get few information about these virtual addresses. Watermarking compels attacker to reverse hash value and watermarked branch. The difficulty of tampering attack is equal to reverse hash value and neural network.

6 Evaluation

In this section we present our evaluation on C/C++ code implementations of the semantic-integrated watermarking. We have measured the cost of embedding the watermark in terms of the time and space overhead incurred by the inserted watermark code and the resiliency of the watermark to attacks. The following attack methods have being considered:

- attackers disassemble the program to locate the watermark and tamper proofing module.
- attackers deploy automatic analysis tools to analyze the trigger of watermarked branch and correct hash value.
- attackers attempt to transform or remove suspicious code for destroying watermark structure.

6.1 Static disassemble

Once the attacker gets the watermarked program, he may first disassemble the program to comprehend the internal logic. When attacker disassembles the program, he can get the following assemble code:

```
var_4= dword ptr −4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push     ebp
mov      ebp, esp
push     ecx
lea      eax, [ebp+var_4]
push     eax
push     offset Format    ; "%d"
call     ds:scanf
add      esp, 8
mov      eax, [ebp+var_4]
call     nullsub_1
call     Update
call     Predict
jmp      eax
_main endp
```

Figure 7 gives the control flow by IDA, we can see IDA cannot collect all paths in main function, in which the `function_a` and `function_b` are invisible in control flow graph. The information attacker can only get is that the function `Predict` and uncondition transfer instruction `jmp eax`. Attacker cannot find any suspicious code except the neural network module. Watermark structure has been embedded in neural network so that watermark path cannot be revealed by static control flow analysis. If attacker attempts to attack watermark, he must analyze the neural network module.

6.2 Remove the watermark and tamper proofing

If the attacker realizes that neural network modules are related with the watermark and correct hash value, he may try to remove the watermark structure from the neural network

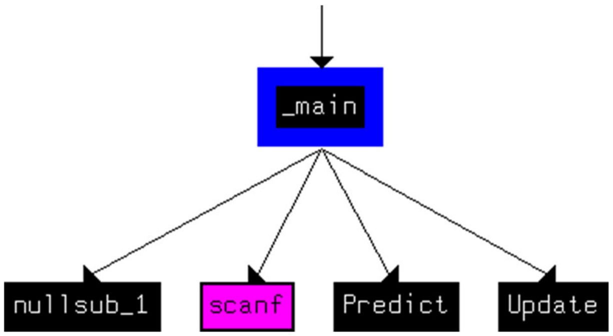


Fig. 7 The control flow of watermarked branch

module. Firstly, attacker want to analysis which input triggers the watermarked path and what is the correct hash value. According to the Ma [32], neural network can resist powerful reverse engineering: concolic execution. Base on this advantage, we have tested a simple condition branch in Section 4.2 to show whether Bitblaze can solve the secret key and hash value.

In experiment, TEMU model has recorded execution trace of test branch. And then we use VINE to compile the trace to intermediate language. Finally, constraint solver will solve the result. Table 2 has given the result of constraint solver. The test input is 7. According to the result, the complexity of watermarked branch is more complex than original branch. The constraint solver cannot get the value which can trigger the watermarked path. It should be emphasized that the experiment only test a simple branch. Analyzing the watermarked branch is beyond ability of existing analysis tools. In a real world application, analyzing whole code blocks of watermarked program will meet more difficulties.

In addition, we evaluate semantic-integrated watermarking using dynamic taint analysis of Bitblaze. We taint the original input x and analyze the taint propagation. All the main added variables are tainted. That is to say, the dynamic analysis tool cannot distinguish the watermarking and program semantics. These results show that our watermark structure is highly integrated with program semantics. Therefore, even the attacker can point the location of watermark, he still cannot remove the watermark. In other words, the price of removing watermark is expensive enough.

6.3 Code transformation

A simple effective attack to watermarking is distortive attack, also call transform attack. This attack chooses code obfuscator, code compression and code packer to make a semantic-preserve transformation to watermarked program. Some variable or internal structure of program changed under distortive attack, which destroys the watermarking recognition. We give the security analysis to distortive attack in Section 5. Now we choose some code

Table 2 The result of constraint solver on watermarked branch

Program	Instruction	Control transfer	STP size	STP result	Constraint
P	95	21	244458B	0×32	19
P_w	426	123	1097886B	N/A	117

Table 3 The distortive attack to watermarked

Tools	Theory	Crash	Recognition	Tamper response
UPX	real time encryption/decryption	true	false	false
ASPack	junk code insert	true	false	ture
WProtect	virtual machine packer	true	false	true

obfuscators and packers to transform watermarked program, then we test whether we can recognize watermark from transformed program. In Table 3, we give the experiment results for UPX [35], ASPack [3] and WProtect. We choose three commonly used code protection tools to transform the watermarked program. Without exception, these transformed programs all crash in execution; some of them lead to respond tampering.

6.4 Performance overhead

In order to assess the practicability of the watermarking, we deploy our watermarking in six benchmarks from SPECint-2006: 999.specrand, 401.bzip2, 429.mcf, 456.hmmer, 458.sjeng and 470.lbm. The environment of experiment is that operating system: Windows7 32-bit, hardware platform: Intel(R) Core(TM) i7-2600 CPU 3.40GHz with 2.00GB RAM).

This paper evaluates our watermarking based on time cost and space cost. The neural network model is trained by Matlab 2012, and its topological structure includes one input layer, two hidden layers (include 7 nodes) and one output layer. Training process is optimized by PSO algorithm. We insert 128 bits watermarking in 16 branches in each benchmark. Table 4 shows the time cost of our watermark. The extra overhead is relative with the execution frequency of watermarked branches. The benchmark 999.specrand increases the 62.5% overhead than original, because the only two branches in program are in a large loop. Except specrand, other benchmarks only increase 2.6% to 12.1% overhead. The computational complexity of watermarked branches is equal to the neural networks; the time complexity of three layers neural network is $O(n^2)$ for predicting one samples. We can conclude that our watermark has affected the execution efficiency in a small range. The run time of each watermarked branch is less than magnitude of $10^{-3}s$. The experimental results show a positive correlation between time cost and watermarked branch execute times. In total, time cost is in the acceptable range.

Space cost is another indicator for evaluation. It indicates the affection that the extra space watermarking occupy. Table 5 shows the program size changed after being watermarked. From the Table 5, the extra space cost for each benchmark is approximately 8KB. The space cost mainly caused by storing the weight matrix of neural network. Meanwhile,

Table 4 The time cost

Program	Origin	Watermarked	Execute times	Overhead
specrand	20.761s	33.527s	48481	62.5%
bzip2	17.328s	19.483s	268	12.1%
mcf	5.715s	5.938s	710	3.4%
hmmer	25.233s	27.178s	35	7.5%
sjeng	15.367s	16.355s	268	6.5%
lbm	3.775s	3.926s	272	2.6%

Table 5 The space cost(unit:byte)

Program	Original	Watermarked	Cost
specrand	32256	40762	8506
bzip2	178176	186296	8120
mcf	49664	57224	7560
hmmer	514048	522130	8082
sjeng	269312	277684	8372
lbm	58880	66102	7222

the oblivious hash spend a certain place in watermarked program. For inserting a fixed length watermarking, the space cost is constant.

7 Related work

7.1 Software watermarking

The watermarking is like a *intellectual property notice*; it can claim the intellectual property ownership to the programs, images or other types of file [4, 23, 26, 31, 34]. According to the embed mode, watermarking can be divided into *static watermarking* and *dynamic watermarking*. Most of the previous works focus on static watermarking [33]. Cousot [19] proposed an abstract interpretation-based watermarking by inserting congruent integers in JAVA code; these big integers which do not often appear in common program may cause the attacker suspicions. Microsoft [20] proposed a watermark by reordering basic blocks. Regrettably, this watermarking can be easily destroyed by simply transform, such as inserting bogus code or obfuscating the program. These seemingly clever static watermarking design still had congenital defects. The stealth is "Achilles Heel" of the static watermarking. Although dynamic watermarking solve this problem in static semantic environment, with the development of software testing, the same problem still exist in dynamic semantic environment. A classical dynamic watermarking is Easter Egg Watermarks [45]. The main problem with Easter Egg watermarks is that they seem to be easy to locate. Watermark structure executes only when the right input sequence has been discovered. Therefore, watermark structure can be easily removed or destroyed and program correctness has not been affected. Based on the design of Easter Egg watermarks, researchers had proposed several dynamic watermarking schemes. One kind is data structure watermarking [14, 18, 29]. Unfortunately, data structure watermarks cannot resist against distortive attack. Because these data structure often store in program by using point operations, several obfuscating transformations can destroy the dynamic data structure and make watermark recognition impossible. Another kind of dynamic watermarking is path-based watermarking, whose idea is to embed condition branch structure into the program [17]. Based on such idea, several researchers have also proposed other similar schemes for execution trace watermarking. However, these watermarking still cannot work under distortive attack.

7.2 Control flow obfuscation

Control flow obfuscation is one of the major techniques of code obfuscation. The purpose of control flow obfuscation is that make the program logic difficult to be understood. Sharif et al. obfuscated `if-else` branch by using hash function and self-decrypting code [38]. The limitation of this work is that the application range is only to protect equal conditions(i.e. `if(x=5)`). If obfuscating unequal conditions(i.e. `if(x>5)`), it may cause too much time cost and space cost in deploying hash table. Wang et al. proposed a linear obfuscation scheme to combat symbolic execution [43]. Wang used the unsolved mathematical problems "Collazt conjecture" to encode branch inputs, so symbolic execution and constraint solver hardly to solve the origin inputs.

Ma [32] and Zong [49] introduced classification algorithms to obfuscate program branches. Different from common use [9–12, 48], the neural network and SVM are used in the opposite direction. Although there are differences between the algorithms to choose, these two obfuscator had the same principles: both neural network and SVM had complex internal logic. This feature of neural network and SVM can help obfuscator resisting against common types of attack. However, being limited by properties of neural network and SVM, obfuscating non-linear branch conditions(i.e. `if(`$x > 5$`)&&(`$y > 5$`))` will cause much time cost and space cost.

7.3 Software tamper proofing

Software tamper proofing is to prevent the program be illegally modified through software or hardware measures. Tamper proofing has these features:(i) it can detect tampering in program; (ii) it can respond tampering by terminate, delete software or output invalid result [13, 20]. A whole tamper proofing techniques contains integrity detection and tamper response. Existing integrity detection methods mainly use check-sum or hash value to tag the integrity of code block. Aucsmith D [5] present a integrity verification kernel(IVK) method based on self-en/decryption and self-modification. IVK only decrypt code block when code block runs and its check sum is correct. The fine-grained partition of code blocks will introduce expensive overhead. Oblivious Hash [15] and its applications [27] provide robust and stealthy detect methods in tamper proofing, and transplant OH to JAVA bytecode. The overhead of deploying OH depends on the number of OH insert; more OH deployment increases the level of security and causes more overhead.

The study of tamper response is relatively less than integrity detection. A typical method is presented by Jin [28]. When it detect tampering, response module set a global pointer to NULL to cause program crash. The limitation of NULL pointer response is that global variable do not exist in JAVA. Besides, with the development of software analysis, this suspicious operation can be easily locate and remove by binary code analysis [46]. Researching the more stealthy respond method is the mainstream in academe.

7.4 Symbolic execution and its application

Symbolic execution has been used in software testing for three decades [8, 30]. It can solve the inputs which trigger the specific execution path. After more than 30 years of development, modern symbolic execution tools can automatically generate test cases, and can cover as many as possible execution paths [7, 21]. Researchers had developed several powerful tools based on symbolic execution. Brumley et al. developed a tools called Mine-Sweeper to detect the trigger condition of malware [6]. Wang et al. removed malicious hidden-code in malware by using symbolic execution and taint analysis [44]. Zeng et al. realized obfuscated code reuse through dynamic symbolic execution [47]. This efficient reverse analysis had caused serious damage to software intellectual property.

Although such techniques are powerful, some limitations still exist [36]. When path constrains cannot reduce to a certain symbol expression, analyzing such code was beyond ability of symbolic execution.

8 Conclusion

Independent semantics, weak strength and weak resilience to state-of-the-art reverse engineering are the involved defects of existing software watermarking. In this paper, we propose a novel software watermarking : semantic-integrated watermarking. We have integrated watermarking and tamper-proofing with program semantics by using neural network. The difficulty of reversing watermarked program is equal to extract rules from neural network and reverse the hash function. The experimental results show that semantic-integrated watermarking could effectively resist the state-of-the-art reverse engineering, yet bring acceptable overhead.

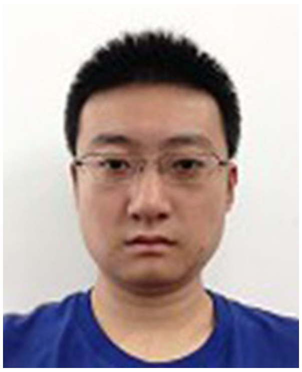
The two limitations of our watermarking are implied in Sections 6.2 and 6.4. It is incompatible with some code protection methods, which caused by the way of watermarking embedding. For a mature process of software development, it need to consider combine with more suitable protection methods. Besides, watermarking need a little more space cost than the existing techniques.

References

1. Akhunzada A, Sookhak M, Anuar NB, Gani A, Ahmed E, Shiraz M, Furnell S, Hayat A, Khan MK (2015) Man-at-the-end attacks: analysis, taxonomy, human aspects, motivation and future directions. J Netw Comput Appl 48:44–57
2. Alliance BS (2016) Bsa global software survey, available link: http://globalstudy.bsa.org/2016/index.html
3. Aspack: executable packer, http://www.aspack.com
4. Atawneh S, Almomani A, Al Bazar H, Sumari P, Gupta B (2017) Secure and imperceptible digital image steganographic algorithm based on diamond encoding in dwt domain. Multimedia Tools and Applications 76(18):18451–18472
5. Aucsmith D (1996) Tamper resistant software: an implementation. In: Information hiding. Springer, pp 317–333
6. Brumley D, Hartwig C, Liang Z, Newsome J, Song D, Yin H (2008) Automatically identifying trigger-based behavior in malware. Botnet Detection 36(1):65–88
7. Bugrara S, Engler D (2013) Redundant state detection for dynamic symbolic execution. In: Proceedings of the 2013 USENIX conference on annual technical conference, USENIX Association, pp 199–212
8. Cadar C, Sen K (2013) Symbolic execution for software testing: three decades later. Commun ACM 56(2):82–90
9. Chang X, Nie F, Wang S, Yang Y, Zhou X, Zhang C (2016) Compound rank-k projections for bilinear analysis. IEEE Transactions on Neural Networks and Learning Systems 27(7):1502–1513
10. Chang X, Ma Z, Lin M, Yang Y, Hauptmann A (2017) Feature interaction augmented sparse learning for fast kinect motion detection. IEEE Trans Image Process 26(8):3911–3920
11. Chang X, Ma Z, Yang Y, Zeng Z, Hauptmann AG (2017) Bi-level semantic representation analysis for multimedia event detection. IEEE Transactions on Cybernetics 47(5):1180–1197
12. Chang X, Yu Y-L, Yang Y, Xing EP (2017) Semantic pooling for complex event analysis in untrimmed videos. IEEE Trans Pattern Anal Mach Intell 39(8):1617–1632
13. Chen H-Y, Hou T-W, Lin C-L (2007) Tamper-proofing basis path by using oblivious hashing on java. ACM Sigplan Notices 42(2):9–16
14. Chen X, Fang D, Shen J, Chen F, Wang W, He L (2009) A dynamic graph watermark scheme of tamper resistance. In: Fifth international conference on information assurance and security, 2009. IAS'09, vol 1. IEEE, pp 3–6

15. Chen Y, Venkatesan R, Cary M, Pang R, Sinha S, Jakubowski MH (2002) Oblivious hashing: a stealthy software integrity verification primitive. In: International workshop on information hiding. Springer, pp 400–414
16. Collberg C, Thomborson C (1999) Software watermarking: models and dynamic embeddings. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, pp 311–324
17. Collberg C, Carter E, Debray S, Huntwork A, Kececioglu J, Linn C, Stepp M (2004) Dynamic path-based software watermarking. ACM Sigplan Notices 39(6):107–118
18. Collberg CS, Thomborson C, Townsend GM (2007) Dynamic graph-based software fingerprinting. ACM Trans Program Lang Syst (TOPLAS) 29(6):35
19. Cousot P, Cousot R (2004) An abstract interpretation-based framework for software watermarking. In: ACM SIGPLAN Notices, vol 39. ACM, pp 173–185
20. Davidson RI, Myhrvold N (1996) Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884
21. Godefroid P, Levin MY, Molnar DA et al. (2008) Automated whitebox fuzz testing. In: NDSS, vol 8, pp 151–166
22. Golea M (1996) On the complexity of rule extraction from neural networks and network querying. In: Rule extraction from trained artificial neural networks workshop
23. Gupta B, Agrawal DP, Yamaguchi S (2016) Handbook of research on modern cryptographic solutions for computer and cyber security. IGI Global
24. Gybenko G (1989) Approximation by superposition of sigmoidal functions. Math Control Signals Syst 2(4):303–314
25. Hornik K (1991) Approximation capabilities of multilayer feedforward networks. Neural Netw 4(2):251–257
26. Ibtihal M, Hassan N et al. (2017) Homomorphic encryption as a service for outsourced images in mobile cloud computing environment. International Journal of Cloud Applications and Computing (IJCAC) 7(2):27–40
27. Jacob M, Jakubowski MH, Venkatesan R (2007) Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings. In: Proceedings of the 9th workshop on multimedia & security. ACM, pp 129–140
28. Jin H, Lotspiech J (2003) Proactive software tampering detection. In: ISC. Springer, pp 352–365
29. Kamel I, Albluwi Q (2009) A robust software watermarking for copyright protection. Comput Secur 28(6):395–409
30. King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
31. Li J, Yu C, Gupta B, Ren X (2017) Color image watermarking scheme based on quaternion hadamard transform and schur decomposition. Multimedia Tools and Applications. https://doi.org/10.1007/s11042-017-4452-0
32. Ma H, Ma X, Liu W, Huang Z, Gao D, Jia C (2014) Control flow obfuscation using neural network to fight concolic testing. In: International conference on security and privacy in communication systems. Springer, pp 287–304
33. Myles G, Collberg C (2003) Software watermarking through register allocation: implementation, analysis, and attacks. In: International conference on information security and cryptology. Springer, pp 274–293
34. Nagra J, Collberg C (2009) Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection. Pearson Education
35. Oberhumer M, Molnár L, Reiser JF (2004) The ultimate packer for executables (upx), Published online at http://upx.sourceforge.net/. Last accessed.
36. Qu X, Robinson B (2011) A case study of concolic testing tools and their limitations. In: International symposium on empirical software engineering and measurement (ESEM), 2011. IEEE, pp 117–126
37. Sen K (2007) Concolic testing. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM, pp 571–572
38. Sharif MI, Lanzi A, Giffin JT, Lee W (2008) Impeding malware analysis using conditional code obfuscation. In: NDSS
39. Shi Y et al. (2001) Particle swarm optimization: developments, applications and resources. In: Proceedings of the 2001 congress on evolutionary computation, 2001, vol 1. IEEE, pp 81–86
40. Song D, Brumley D, Yin H, Caballero J, Jager I, Kang M, Liang Z, Newsome J, Poosankam P, Saxena P (2008) Bitblaze: a new approach to computer security via binary analysis. In: 4th International Conference on Information systems security. Springer, pp 1–25
41. Tickle AB, Andrews R, Golea M, Diederich J (1998) 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):1057–1068

42. Towell GG, Shavlik JW (1993) The extraction of refined rules from knowledge-based neural networks. In: Machine learning, Citeseer
43. Wang Z, Ming J, Jia C, Gao D (2011) Linear obfuscation to combat symbolic execution. Computer Security–ESORICS 2011:210–226
44. Wang Z, Jia C-F, Lu K (2012) Malicious hidden-code extracting based on environment-sensitive analysis. Jisuanji Xuebao (Chinese Journal of Computers) 35(4):693–702
45. Wolf D, Wolf A (2003) The easter egg archive. Available online: http://www.eeggs.com [Last Accessed: April 13, 2014]
46. Yu X (2017) The design of branch obfuscation - based dynamic software water- marking. Master's thesis, Nankai University
47. Zeng J, Fu Y, Miller KA, Lin Z, Zhang X, Xu D (2013) Obfuscation resilient binary code reuse through trace-oriented programming. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security. ACM, pp 487–498
48. Zhang Z, Sun R, Zhao C, Wang J, Chang CK, Gupta BB (2017) Cyvod: a novel trinity multimedia social network scheme. Multimedia Tools and Applications 76(18):18513–18529
49. Zong N, Jia C (2014) Branch obfuscation using "black boxes". In: Theoretical aspects of software engineering conference (TASE), 2014. IEEE, pp 114–121



Zhe Chen PHD candidate in Nankai University. His main research interests include code obfuscation and software watermarking £®

Zhi Wang PhD lecturer. His main research interests include code obfuscation, malware analysis and prevention £®

Chunfu Jia PhD, professor and PhD supervisor. His main research interests include network and system security £¬ cryptography application and malware analysis £®