A Buffer Overflow Based Algorithm to Conceal Software Watermarking Trigger Behavior

Jiu-jun CHENG†, Shangce GAO††††, Catherine VAIRAPPAN†††††, Nonmembers, Rong-Long WANG†††††, Member, and Antti YLÅ-JÄÅSKI††††††, Nonmember

SUMMARY Software watermarking is a digital technique used to protect software by embedding some secret information as identification in order to discourage software piracy and unauthorized modification. Watermarking is still a relatively new field and has good potential in protecting software from privacy threats. However, there appears to be a security vulnerability in the watermark trigger behaviour, and has been frequently attacked. By tracing the watermark trigger behaviour, attackers can easily intrude into the software and locate and expose the watermark for modification. In order to address this problem, we propose an algorithm that obscures the watermark trigger behaviour by utilizing buffer overflow. The code of the watermark trigger behaviour is removed from the software product itself, making it more difficult for attackers to trace the software. Experiments show that the new algorithm has promising performance in terms of the imperceptibility of software watermark.

key words: software watermarking, trigger behaviour, buffer overflow

1. Introduction

Software piracy can be defined as the unauthorized use of software, illegal duplication of copyrighted software or the installation of copyrighted software on computers other than those authorized, under terms of the software license agreement. According to the report published by Business Software Alliance (BSA), the rate of global software piracy was 42% in 2011 with a commercial value of $63.4 billion. Just in the last decade, more and more software is being distributed illegally creating a heterogeneous environment that calls for immediate actions to be taken on improving software authorization. Naturally, adequate security is required to protect the software ownership against reverse engineering, illegal tampering, program-based attacks, BORE (Break Once Run Everywhere) [1] and unauthorized use of software. Efforts on improving the mechanism on software authorization has been done using cryptographic techniques, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), anti-virus programs, firewalls, etc [2].

One of the more commonly used methods to stop software piracy is software ownership authentication method through watermark embedding. Watermark embedding is a technique that embeds a secret message into the software hosts to determine ownership of code and helps in preventing tampering of source code. Some of the more recent works have developed techniques for embedding watermarks into various other medias, such as images [3]–[5], audio [6] or video files [7]. Other works include those by Collberg, Palsbery etc. [8]–[10] and patents [11]–[13] show how watermarks such as copyright notices, can be embedded into software programs.

Static software watermarks are stored in the application executable itself either by data or code and it can be detected without running the program. One of the earliest patent for software watermark algorithm was introduced by Davidson and Myhrvold [14], which embeds the watermark by reordering the basic blocks of a control flow graph. Based on this idea, first graph-based software watermarking algorithm through the insertion of a directed sub graph known as WS or GTW was proposed [15]. Collberg et al. [16] modified the implementation of GTW, known as GTWsm by encoding the watermark as a reducible permutation graph (RPG) [17], a reducible control flow graph [18], [19] with maximum out-degree of two, mimicking a real code. Although static representation is an inexpensive detection process, it is very susceptible for attack mainly due to the simplistic nature of it.

On the other hand, dynamic software watermarks [20], [21] are dynamically produced during software running, and it has better concealment characteristic. It has made a considerable progress and even became a popular technique for copyright protection of multimedia information [22]. The watermark content is concealed within the data structure of software running space, and the concealed content can also be developed with some dynamical diagram of topological structure from static value [15], [23]. Researches have proposed algorithm with robust data structure that can be used to encode watermark information, for example, Parent-Pointer Trees, Planted Planar Cubic Trees (PPCT) [24], List-type Graphs, Reducible Permutations Graphs (RPG), and so on. Purdue University has developed an experimental dynamic diagram watermark system which is called JavaWiz [10], and it is an embedded type of PPCT tree structure coded in Java source code.
Dynamic watermark technology actually adopts “user interactions” as the watermark trigger behaviour [25] to ensure concealment of content. However, once the watermark trigger behaviour is attacked, it is then very easy to gain access to the whole process by tracing the program executing code. Validation of the behaviour is shown in the code and through reversal engineering methodology, one can easily trace the process to understand the trigger behaviour and its producing process. The more commonly available user interactions are input devices such as keyboard and mouse. For example, in PhotoShop application, the watermark can be easily created by pressing “Ctrl” and “Alt” while clicking the “about PhotoShop ...” menu. But the question remains if such measures are sufficient to prevent software plagiarism and piracy? On the other hand, there are still inadequate research on the trigger behaviour and realizing the need for a higher protection against attacks.

In conventional watermarks, the generation, display and trigger behaviors of the watermark are contained in the software product itself. The watermark is triggered by the user interactions. The safety of watermarks strongly depends on the concealment of the watermark trigger behavior. Once the trigger behavior is revealed to the software attacker, the watermark is easily to be destroyed. Although in most cases, attackers cannot obtain such trigger behavior in a simple way, they are still able to locate the source code of triggering behaviors by using reverse engineering approaches. The reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction, and it is usually used to create unlicensed or unapproved duplicates. Even there is no source code available for the software, any efforts towards discovering one possible source code for the software are regarded as reverse engineering. Generally, it can be accomplished by either a disassembly using disassemblers or a decompilation using decompilers. The issues of how to implement the reverse engineering are out of the discussion of this work, interested readers can refer the Refs. [26], [27] for details. As a result, attackers are able to track and modify the watermark after locating the source code of trigger behavior based on reverse engineering approaches, even though the efforts of carrying out such approaches might be somewhat expensive. In an extreme case, if the watermark trigger behavior is exposed and divulged to attackers accidentally rather than programmed located, the watermark to protect software copyright will be easily modified, thus causing software plagiarism and piracy.

To address the above issues, the utilization of buffer overflow to conceal the watermark trigger behavior is proposed in this paper for the first time, because the concealment of watermark trigger behavior using user interactions with the software itself has potential risk of being destroyed. By making use of the bilayer technique of restoration to deal with system stack and the shellcode deployment to manipulate the predefined software behaviors, the generation and trigger of the watermark can be accomplished. As long as the shellcode’s address and the decryption key which are preserved by the software owner are kept secret from attackers, the proposed approach is able to delete the validation of trigger behaviour from software codes, thus protecting the software from privacy and other security risk. It is worth emphasizing that the only difference between the proposed buffer overflow based approach and the conventional static or dynamic approaches is the method of triggering the watermark, while the watermark itself is still embedded in the target program.

The rest of the paper is organized as follows. In the next section the main idea on how a software watermark trigger works is presented. The details of the proposed algorithm is discussed in Sect. 3, and the experimental results for the implementation of the proposed algorithm can be found in Sect. 4. We concluded the paper with a short summary of the proposed algorithm and its performance in the last section.

2. The Trigger Behaviour of Software Operation

To make the paper self-explanatory, the software execution process is briefly introduced to make readers a better understanding on the details of watermark trigger behaviour. Thereafter, the call function together with the buffer overflow are highlighted within the software execution framework, aiming to explain the effects of call function and the reasons of choosing buffer overflow as a watermarking trigger.

2.1 Software Execution Process

When a software runs, the code and data are firstly loaded into memory from files. In Win32 platform for example, the format of PE (PreInstallation Environment) file in disk file is similar with memory space. It can be conveniently mapped from file to virtual memory. PE file begins with offset address of zero and copies each block of file into memory space. Once the work initializer has been completed, the running software starts with entry point which is defined by PE file header and assigns the outset address to the instruction register EIP (Extended Instructions Pointer). When the program runs in the memory, the EIP instruction register will direct the outset address of next instruction. The software behaviour is actually controlled by EIP. Beginning with entry point of program, the next instruction address execution process are as shown in Fig. 1.

Based on the type of the current executing instruction, there are four cases in which the instruction register EIP achieves the next instruction address.

- Sequence execution: The EIP directs the address of instruction A, known as addr1. When it begins to execute instruction A, EIP automatically directs the address of the next instruction B, addr2. The address of instruction B equals the address of instruction A which includes the length of instruction A as well and it is automatically realized by accumulator.
### Table 1 The main operation on call function.

| Assembly instruction | Descriptions | Code position |
|-----------------------|--------------|---------------|
| PUSH arg2            | the parameters will be pushed into the stack according to the order from right to left. | father function |
| PUSH arg1            |                          |               |
| CALL Function1        | return address will be pushed into the stack, and go to the entry point of sub-function. |               |
| PUSH EBP             | save the pointer on the stack frame and the bottom stack of father function. | sub-function |
| MOV EBP, ESP         | set the pointer on the stack frame and the bottom stack of sub-function. |               |
| SUB ESP, XXX         | raise the top stack in order to open up space for stack frame of sub-function. |               |

### Table 2 The main operation on return function.

| Assembly instruction | Descriptions | Code position |
|-----------------------|--------------|---------------|
| ADD ESP, XXX          | recycle stack frame space of sub-function. | sub-function |
| POP EBP               | restore the pointer on stack frame and bottom stack of father function. |               |
| RET                   | return address pops up, and goes to return address function. |               |
| ADD ESP, 08           | recycle two parameters space in stack. | father function |

### Fig. 1 Four cases of EIP instruction address.

- **Goto instruction:** The EIP directs the address of instruction A, called addr3. The instruction A is a Goto instruction (conditional Goto or unconditional Goto), and the Goto address of the next instruction B is given in the operand of instruction A, called addr4. After instruction A has been executed, EIP would direct the Goto destination address, that is addr4.
- **Function call instruction:** The EIP directs the address of instruction A, called addr5. The instruction A is a function call instruction (CALL), and the function outset address is given in the operand of instruction A, called addr7. After instruction A has been executed, the address of the next instruction B (also known as return address), that is addr6, will be pushed into the stack. EIP will direct the address of the call function, addr7.
- **Function return instruction:** The EIP directs the address of instruction A, called addr9. Instruction A is a function of return instruction (RET). After instruction A has been executed, the return address, that is addr6, will pop up from the stack. EIP will then direct the function return address, addr6.

By comparing the four kinds of instruction, the address origin for the next instruction are respectively as follows:

- **Goto instruction:** The EIP directs the address of instruction A, called addr3. The instruction A is a Goto instruction (conditional Goto or unconditional Goto), and the Goto address of the next instruction B is given in the operand of instruction A, called addr4. After instruction A has been executed, EIP would direct the Goto destination address, that is addr4.
- **Function call instruction:** The EIP directs the address of instruction A, called addr5. The instruction A is a function call instruction (CALL), and the function outset address is given in the operand of instruction A, called addr7. After instruction A has been executed, the address of the next instruction B (also known as return address), that is addr6, will be pushed into the stack. EIP will direct the address of the call function, addr7.
- **Function return instruction:** The EIP directs the address of instruction A, called addr9. Instruction A is a function of return instruction (RET). After instruction A has been executed, the return address, that is addr6, will pop up from the stack. EIP will then direct the function return address, addr6.

By comparing the four kinds of instruction, the address origin for the next instruction are respectively as follows:

- **Goto instruction:** The EIP directs the address of instruction A, called addr3. The instruction A is a Goto instruction (conditional Goto or unconditional Goto), and the Goto address of the next instruction B is given in the operand of instruction A, called addr4. After instruction A has been executed, EIP would direct the Goto destination address, that is addr4.
- **Function call instruction:** The EIP directs the address of instruction A, called addr5. The instruction A is a function call instruction (CALL), and the function outset address is given in the operand of instruction A, called addr7. After instruction A has been executed, the address of the next instruction B (also known as return address), that is addr6, will be pushed into the stack. EIP will direct the address of the call function, addr7.
- **Function return instruction:** The EIP directs the address of instruction A, called addr9. Instruction A is a function of return instruction (RET). After instruction A has been executed, the return address, that is addr6, will pop up from the stack. EIP will then direct the function return address, addr6.

### Fig. 2 The stack frame operation on call and return function.

- **Software behaviour can be controlled by modifying the address of the return function instruction.** Theoretically, by modifying the address of return function, the program may go to any expected instruction address, and localize the code produced by software watermarking.

Function1 which includes parameters arg1 and arg2 are used to describe the main operation on call and return function by referring to Table 1, 2, and Fig. 2.

From Fig. 2, the stack frame of the current function is located between EBP (Extended Base Pointer) and ESP (Extended Stack Pointer). The position of local variable in stack frame is exactly at the nearby top of return address. If the local variable has buffers like array, the weakness of buffer...
Table 3  Embedding copyrights information in stack overflow.

| Machine code (hex) | Assemble instruction (mnemonic) | Notes                                                                 |
|-------------------|---------------------------------|----------------------------------------------------------------------|
| 33 DB             | XOR EBX, EBX                     | EBX zero clearing                                                    |
| 53                | PUSH EBX                         |                                                                      |
| 68 73 69 74 79    | PUSH 79746973                    |                                                                      |
| 68 69 76 65 72    | PUSH 72657869                    |                                                                      |
| 68 69 20 75 6E    | PUSH 66752069                    |                                                                      |
| 68 6F 6E 67 6A    | PUSH 6A676E6F                    |                                                                      |
| 68 63 63 20 74    | PUSH 74206363                    |                                                                      |
| 68 39 20 68 70    | PUSH 70682039                    |                                                                      |
| 68 29 32 30 30    | PUSH 30303229                    |                                                                      |
| 68 74 20 28 63    | PUSH 63282074                    |                                                                      |
| 68 72 69 67 68    | PUSH 68676972                    |                                                                      |
| 68 43 6F 70 79    | PUSH 79706F43                    |                                                                      |
| 8B C4             | MOV EAX, ESP                      | character string pointer is assigned to EAX.                        |
| 53                | PUSH EBX                         | Four parameters are pushed into system stack                         |
| 53                | PUSH EBX                         | with order from right to left, respectively.                        |
| 50                | PUSH EAX                         | (0, “Copyright (c) 2013 hpc tongji university”,0,0)                 |
| 53                | PUSH EBX                         |                                                                      |
| 8B 02 07 D5 77    | MOV EAX, 0x7D50702               | Call system function Message-BoxA to display character string.      |
| FF D0             | CALL EAX                         |                                                                      |

overflow can be adopted to the return address by the overflow data. It can then go to the address defined by user to execute the code when sub-function returns [28].

Buffer overflow is a most common type of leak for software [29], and it originates from the design flaws of the John Von Nouma’s architecture, which indeed has nothing to do with operating system or software environment itself. However, it is possible to conceal the watermark code well because the entry point of watermark code doesn’t operate in the program. Watermark attackers on the other hand, are unable to find the watermark by tracing the code. Software behaviour usually depends on users’ input since only the copyright owners know the address of watermark code, which they correctly input during the withdrawal.

Due to the defect of buffer overflow, users can input the specific content to localize and trigger the concealed watermark code in software products. Based on this understanding, we are attempted for the first time to utilize the triggering behaviour during the return of the function for the system stack overflow in our proposed algorithm.

3. The Proposed Algorithm

In this section, we firstly introduce the method of generation and display of watermark. The details of how we trigger the watermark are depicted in the following two subsections, where an attempt of utilizing buffer overflow to conceal the watermark trigger behaviour is made.

3.1 The Generation and Display of Watermark

In a computer program, both the code and data are stored in binary format, even though they are loaded into different segments once the software runs. Whether a content of a certain address is bound to either data or code, depends on function instruction and in fact it has nothing to do with address position. Figure 3 describes the relationship between the code and data as it is used in program.

As indicated in Fig. 3, the instruction will put the code content of “mov eax, address1” as data into EAX register although address1 actually stores code. Likewise, for the instruction “jmp address2”, although it stores data, the instruction will take data as code. It will then decode and execute. In most cases, the data segment content can’t be explained based on legal instruction which results in the lead program to collapse.

Using the buffer leak and the flaws that computers can’t differentiate code and data, code can be embedded into data segments of software’s executable file. Table 3 describes a section of Shellcode which displays the copyright information.

In Table 3, the message box for the copyright clarification is set as “Copyright (c) 2013 hpc tongji university” in the program, and it includes the owner information as well. During the watermark development stage, the section of Shellcode is stored in the software, and buffer overflow is used to trigger. As long as it replaces the return function address with the outset address of the Shellcode section, it can then directly localize the watermark code and complete the watermark process.
3.2 System Stack and Shellcode Deployment

As for the buffer overflow, there are three kinds of data which is filled into the stack frame function:

- Invalid Value: It can be any other value except 0x00. In general the value of 0x90 which responds with NOP (No Operation) instruction will be selected to fill in order to make sure that it doesn’t affect the program logic.

- Return Value: Return address is filled with 32 bits of new address, it can point to Shellcode’s outset position or the address of indirect Goto instruction like “JMP ESP”.

- Shellcode: This is the executable binary code that can be used to complete the predefined software behaviour, e.g. used for all kinds of initialization, Goto or decrypt instruction and so on.

There are two main kinds of Shellcode deployment in a function stack frame as presented in Fig. 4 (a), (b) [30]. Both methods are suitable in terms of buffer space. In the case of Shellcode being deployed after return address, operation the popping and pushing operation of the stack does not destroy the Shellcode because the stack’s top pointer location at the top. However, it could destroy the preceding stack frame at a wider range. Normally the indirect localization over ESP (as shown in Fig. 4 (c)) is selected due to the limitation of local variable buffer space which is sometimes unable to accommodate several hundreds byte sizes required for a practical Shellcode. In our current work, considering the limited capacity of Shellcode and also the no risk effect to the preceding stack frame, Fig. 4 (b) is used to deploy Shellcode.

3.3 System Stack Restoration

During the operational process of the Shellcode, the preceding EBP stack frame and data on the return address function are destroyed, resulting in the function not being able to return to normal condition. To overcome this, it tries to exit the program using the ExitProcess function but this affects the software operation. As an alternative method, it tries to restore the system stack by using a bilayer function stack frame as described in Fig. 5.

3.3.1 Bilayer Technique of Restoration

As shown in Fig. 5, the operation between inner and outer function is able to restore system stack. It goes to Shellcode outset position based on the return address of inner stack frame function and execute’s the users’ predefined code. The stack frame space of outer function can be used to hold Shellcode and at the same time, it can also protect the bottom pointer of the preceding EBP stack frame and the return address of outer function to return back to normal operation.

Figure 6 describes the three main steps involved in the system stack restoration based on bilayer function stack frame.

- Save EBP register: During execution, the EBP register points to 0x90909090 (which is filled by NOP instruction), the top stack of outer function of stack frame.
Because of the size of the stack frame which has been defined during compilation, executing “mov ebp, esp” and “add ebp, xx” instruction can make the EBP register to point to the bottom stack of the outer stack frame function.

• Restore ESP register: Upon the return of the function, ESP and EBP register both point to the bottom of the current frame stack function. Since the EBP register pointed to the bottom, by executing “mov esp, ebp” instruction, ESP pointer can move to the bottom position of the stack frame. If EBP and ESP register are correctly positioned to the bottom stack of stack frame, it could automatically restore the balance upon the return of the function.

• Restore EBP and EIP register: Since the preceding EBP stack frame and the return address is in good condition and not damaged as in the situation of buffer overflow, by executing “pop ebp” instruction, EBP points to the bottom stack of the call function. The execution of “ret” instruction results in the transmission of return address of EIP, in which Shellcode also correctly positions itself by the outer function.

3.3.2 Bottom Stack Pointer Restoration

As indicated in Fig. 6, the address difference between EBP and ESP as computed in the first step of “add ebp, xx” instruction is one of the biggest disadvantage. The codes shown in Fig. 7 are the operation for function stack frame when it enters into the outer function, Function1.

There are three main steps that are involved for the stack frame operation in the outer function, Function1 before the inner Function2 is called.

• Raise the top stack: VC compiler will by default raise 64 bits (0x40) for the top stack and clear up the stack space. The bottom stack is the area for the local variable function and the variable isn’t defined in Function1. Therefore the length of local variable area is zero.

• Register push: VC compiler will push EBX, ESI, ...
### Table 4  Watermark generation and display in data.

| Machine code (hex) | Assembly instruction (mnemonic) | Notes |
|--------------------|---------------------------------|-------|
| 90                 | NOP                             | Self-defined outset mark |
| 8B 8B              | MOV EBP,ESP                     | ESP+50 is assigned to EBP, points to the bottom stack of current stack frame function. |
| 83 C5 50           | ADD EBP,50                      | code in table 3 |
| ...                | ...                             |       |
| 8B E5              | MOV ESP,EBP                     | recover ESP register |
| 5D                 | POP EBP                         | withdraw Shellcode, turn to call function. |
| C3                 | RETN                            |       |

### Table 5  Data input in the buffer overflow.

| Original content of stack frame | Buffer overflow input | Descriptions                             |
|---------------------------------|-----------------------|------------------------------------------|
| Local variable array            | 90 90 90 90 ...      | filled with NOP                          |
| preceding stack frame EBP      | 90 90 90 90          |                                          |
| return address                  | 77 DC 27 40          | the address of “JMP ESP” instruction in program space. |
|                                 | B8 20 30 50 40       | EAX=00405030, the outset address of Shellcode in data segment. |
|                                 | C1 E8 08             | SHREAX,8                                 |
|                                 | 33 C9                | XOR ECX,ECX                              |
|                                 | B1 70                | MOV CL,70                                |
|                                 | B3 44                | MOV BL,44                                |
|                                 | 30 IC 08             | XOR BYTE PTR DS:[EAX+ECX], BL.           |
|                                 | E2 FB                | LOOPD SHORT 00401020                     |
|                                 | 50                   | PUSH EAX                                 |
|                                 | C3                   | RETN                                     |

$\text{0xCC}$ is filled in turn with parameters EDI, ESI, EBX and when the stack top is raised, and the total space is 80 bits.

4.1 Watermark Generation and Display

The buffer overflow based watermarking generally consists of three steps, i.e., software development, software release, and watermark validation. Software development is responsible for two main tasks, that is, the watermark generation including the exhibition of message box, and the restoration of compiled codes in data segments. It is worth pointing out that the return address is reserved in user guide rather than in the software, thus preventing attackers to track the watermark using reverse engineering. Software release distributes the user guide including watermark declaration to software users.

In the simulation, the codes presented in Table 3 are for the display of the copyright information and with a little more additional instructions, we are able to maintain the system stack as described in Table 4.

The XOR computing encryption technique using 0x44 as the encryption key is written into the software data segment. It inserts garbage code prior to the encrypted code in order to conceal data and improve code security (Table 4).

4.2 Watermark Trigger

Watermark validation is used to prove the copyright of the software owner. It generally uses three procedures: first, construct the buffer overflow for specific user input; then, relocate the return address of the watermark; and finally implement the watermark generation codes to show the software ownership. It should be noted that the implementation codes of latter two procedures in watermark validation are not contained in the software, but in the buffer overflow, thus difficult to be traced by attackers.

In the experiment, bilayer function technique involving Function1 and Function2 has been adopted to return to the normal operation of the buffer overflow. The buffer overflow is carried in the inner function, Function2 and in parallel with the decryption of the code which contains the watermark content.

Referring to Table 5, during the buffer overflow, the Shellcode’s address and the decryption key must be defined during the input of the buffer overflow process so that the copyright information can be displayed during the decryption and localization. Since the key address cannot be displayed in the software code, attacker will find it difficult to trace the watermark trigger behaviour. Implementation of this proposed technique can protect software from software privacy and other security risk.

Once the software watermark code has been triggered and executes successfully, it will then display a prompt dialogue box which contains the software’s copyright information, e.g. “Copyright (c) 2013 hpcc tongji university” as shown in Fig. 9.
4.3 Remarks

The distinct property of the proposed algorithm is the concealment of trigger behaviors using buffer overflow as illustrated in Table 6. The watermark generation and display are observable for both software owners and attackers. However, as the shellcode’s address and decryption key are contained in the user guide, unobservable for attackers, thus effectively preventing attackers to trace and modify the watermark.

To conclude, the proposed buffer overflow based algorithm enables the watermark to be better concealed by completely eradicating the direct relationship between watermark and its trigger behaviors. Furthermore, the watermark can also be concealed in any desired place, exhibiting good flexibility, and it is only located and implemented by trigger behaviors out of the software (i.e. through buffer overflow).

On the other hand, the proposed algorithm also has inherent security vulnerability if the leak of the buffer overflow is utilized by attackers. In addition, the development of the proposed algorithm is somewhat difficult, requiring the watermark developer be familiar with the reverse engineering technique.

5. Conclusion

In this paper, we proposed a new algorithm to conceal the software watermark trigger behaviour. To the best of our knowledge, this is the first work to utilize the buffer overflow to conceal the watermark trigger behaviour.

One of the advantages of this algorithm is the simplicity of its construction process. First, a short segment of assembly code which displays a character string of the copyright information such as “Copyright (c) 2013 hpcc tongji university” is constructed and concealed in the software product that is to be distributed. Then, a buffer overflow is generated in order to localize the watermark according to the Shellcode input by the user. Finally, the watermark will be executed and pop up message box will appear with the copyright information. This newly proposed algorithm scheme is invisible and does not reveal any information about the watermark or its location in the program. Since the zero-knowledge proof is independent of the encoding, it is would be difficult for the attacker to write a legal Shellcode to localize the watermark accurately or even to trace the trigger behaviour. The experimental results indicate the robustness of this technique especially in improvement of copyright protection and ownership identification.

Acknowledgement

This work is partially supported by the National Natural Science Foundation of China under Grants 61203325, 61003205, 61003102 and 61271114, Genguang Project of Shanghai Educational Development Foundation (No. 12CG35), the Fundamental Research Funds for the Central Universities (No.0800219164 and 13D110416), National Science and Technology Support Plan (Grant No. 2012BAH15F03), and Natural Science Foundation Programs of Shanghai (No. 13ZR1443100) respectively.

References

[1] A.T. Inc., “Protecting .net software applications, arxan best practices white paper.” http://www.rcp buyersguide.com/dload.php?file= whitepapers/SponsorIndex_Arxan_Whitepaper24947759.pdf.
[2] A. Ghafoor and S. Muftic, “Cryptonet: Software protection and secure execution environment,” Int. J. Computer Science and Network Security, vol.10, no.2, pp.19–26, 2010.
[3] C.F. Wu and W.S. Hsieh, “Image refining technique using digital watermarking,” IEEE Trans. Consum. Electron., vol.46, no.1, pp.1–5, 2000.
[4] D. Kahn, “The history of steganography,” Lect. Notes Comput. Sci., vol.1174, pp.1–5, 1996.
[5] P.W. Wong and N. Memon, “Secret and public key image watermarking schemes for image authentication and ownership verification,” IEEE Trans. Image Process., vol.10, no.10, pp.1593–1601, 2001.
[6] D. Kirovski and H. Malvar, “Robust spread-spectrum audio watermarking,” IEEE International Conference on Acoustics, Speech, and Signal Processing, pp.1345–1348, 2001.
[7] F. Hartung and B. Girod, “Digital watermarking of uncompressed and compressed video,” Signal Process., vol.66, no.3, pp.283–301, 1998.
[8] C. Collberg and C. Thomborson, “Software watermarking: Models and dynamic embeddings,” Proc. 26th Annual SIGPLAN-AIGACT Symposium on Principles of Programming Languages, pp.311–324, 1999.
[9] C. Christian, C.C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Tech. Rep. 1173-3500, Department of Computer Science, The University of Auckland, New Zealand, 1997.
[10] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, S. Qiuyun, and Y. Zhang, “Experience with software watermarking,” Proc. 16th IEEE Annual Conference on Computer Security Applications, pp.308–316, 2000.
[11] K. Holmes, “Computer software protection,” US Patent 5,287,407, International Business Machines, 1994.
[12] S.A. Moskowitz and M. Cooperman, “Method for steganographic protection of computer code,” US Patent 5,745,569, The Dice Company, 1996.
[13] P.R. Samson, “Apparatus and method for serializing and validating copies of computer software,” US Patent 5,287,408, Autodesk Inc., 1994.
[14] R. Davidson and N. Myhrvold, “Method and system for generating and auditing a signature for a computer program,” US Patent 5,559,884, Microsoft Corporation, 1996.
[15] R. Venkatesan, V. Vazirani, and S. Sinha, “A graph theoretic approach to software watermarking,” Lect. Notes Comput. Sci., pp.157–168, 2001.
[16] C. Collberg, A. Huntwork, E. Carter, G. Townsend, and M. Stepp, “More on graph theoretic software watermarks: Implementation, analysis, and attacks,” Information and Software Technology,
sensor network.

[17] C. Collberg, S. Kobourov, E. Carter, and C. Thomborson, "Error-correcting graphs for software watermarking," Proc. 29th Workshop on Graph Theoretic Concepts in Computer Science, pp.156–167, 2003.

[18] M. Hecht and J. Ullman, “Flow graph reducibility,” Proc. Fourth Annual ACM Symposium on Theory of Computing, pp.238–250, 1972.

[19] M. Hecht and J. Ullman, “Characterizations of reducible flow graphs,” J. ACM (JACM), vol.21, no.3, pp.367–375, 1974.

[20] C. Collberg and E. Carter, “Dynamic path-based software watermarking,” Proc. ACM SIGPLAN 2004 conference on Programming language design and implementation, pp.107–118, 2004.

[21] M. Zhang and G. Xu, “Dynamic data flow graph-based software watermarking,” IEEE International Conference on Network Infrastructure and Digital Content, pp.1029–1033, 2009.

[22] I. Cox, J. Kilian, T. Leighton, and T. Shamon, “A secure, robust watermark for multimedia,” in Information Hiding, pp.185–206, Springer, 1996.

[23] C. Collberg, A. Huntwork, and E. Carter, “Graph theoretic software watermarks: Implementation, analysis and attacks,” Proc. 6th Workshop on Information Hiding, pp.192–207, 2004.

[24] J. Chen, "Dynamic map based on ppct structure software watermark protection," World Automation Congress (WAC), pp.133–136, 2012.

[25] S.J.H. Zaidi, “On the analysis of software watermarking,” 2nd International Conference on Software Technology and Engineering (ICSTE), pp.26–30, 2010.

[26] E.J. Chikofsky and J.H. Cross, “Reverse engineering and design recovery: A taxonomy,” IEEE Softw., vol.7, no.1, pp.13–17, 1990.

[27] C. Peikari and A. Chuvakin, Security warrior, O’Reilly, 2009.

[28] O. Aleph, “Smashing the stack for fun and profit,” Phrack Magazine, pp.7–49, 1996.

[29] F. Gadaleta, Y. Younan, and B. Jacobs, “Instruction-level countermeasures against stack-based buffer overflow attacks,” Proc. 1st EuroSys Workshop on Virtualization Technology for Dependable Systems, pp.7–12, 2009.

[30] O. Failwest, Security: software leak analysis technique, Electronic Industry Press, Beijing, 2008.

Jiu-jun Cheng obtained his PhD degree from Beijing University of Posts and Telecommunications in 2006. He is presently an associate professor of Tongji University. He received the support from Research Fund for the Doctoral Program of Higher Education and Young Excellent Talents in Tongji University. He is author of more than 30 conference and journal papers. His research interests span the area of P2P Overlay and Social networks with a focus on IN/Internet interworking, information security, and wireless systems.

Shangce Gao received the B.S. degree from Southeast University, Nanjing, China in 2005, and the M.S. and PhD. degrees in intellectual information systems and Innovative life science from University of Toyama, Toyama, Japan in 2008 and 2011, respectively. From 2011 to 2012, he was an associate research professor at the Key Laboratory of Embedded System and Service Computing, Ministry of Education, Tongji University, Shanghai 200092, China. He has received the Outstanding Academic Performance Award of IEICE Hokuriku Branch in 2008, the Outstanding Academic Achievement Award of IPSJ Hokuriku Branch in 2011. He is currently an associate professor at College of Information Sciences and Technology, Donghua University, Shanghai, 201620, China. His main research interests are intelligent computing and information processing.

Rong-Long Wang received the B.S. degree from Hangzhou teacher’s college, Zhejiang, China and an M.S. degree from Liaoning University, Liaoning, China in 1987 and 1990, respectively. He received his D.E. degree from Toyama University, Toyama, Japan in 2003. From 1990 to 1998, he was an Instructor in Benxi University, Liaoning, China. In 2003, he joined University of Fukui, Fukui Japan, where he is currently an associate professor in Department of Electrical and Electronics Engineering. His current research interests include soft computing and optimization problems.

Antti Ylä-Jääski is one of the three professors responsible for the Data communications Software the Department of Computer Science and Engineering at Aalto University. His expertise area is network technologies, application and service development in Internet, and network architectures. His current research interests include mobile networks, Internet technologies, service management, and heterogeneous network environments (e.g., 2G/3G, WLAN, PAN, Broadcast).