Proceedings of the IASTED International Conference
Software Engineering and Applications (SEA 2010)
November 8 - 10, 2010 Marina Del Rey, USA

A SOFTWARE PROTECTION METHOD BASED ON TIME-SENSITIVE CODE AND SELF-MODIFICATION MECHANISM

Yuichiro Kanzaki
Dept. of Human-Oriented Information Systems Engineering
Kumamoto National College of Technology
Koshi, Kumamoto, Japan
email: kanzaki@kumamoto-nct.ac.jp

Akito Monden
Graduate School of Information Science
Nara Institute of Science and Technology
Ikoma, Nara, Japan
email: akito-m@is.naist.jp

ABSTRACT

This paper proposes a systematic method for protecting software against malicious reverse engineering attacks. Our method aims to increase the cost of obtaining secret information in a program on the assumption that the adversaries have the ability to perform dynamic analysis as much as static analysis. A program protected by our method contains many time-sensitive codes, which are overwritten with fake (dummy) codes. Each time-sensitive code is modified during execution via self-modification according to the time taken to execute a designated block of the program. If the execution time of the block is within the predetermined range, the time-sensitive code becomes the original one. On the other hand, if the execution time is out of the range, the time-sensitive code becomes the other fake one. In order to obtain the secret information by static analysis, the adversary must find the routines that modify time-sensitive codes which are scattered over the program, and must guess the predetermined valid execution time of the target blocks. In order to obtain the secret information by dynamic analysis, the adversary must make the execution reach the restricted points of the program without stopping the execution. As a result, our method helps to construct highly invulnerable software.

KEY WORDS
software security, software protection, program obfuscation, program camouflage, self-modification

1 Introduction

There has been an increasing need for protecting *secret information* stored in software products, such as algorithms that are commercially valuable, the secret keys for DRM system, and conditional branch instructions for license checking [1]. Since the leakage of secret information causes serious damage to software vendors [2], protecting software against illegal reverse engineering attacks has become an overarching issue.

Many methods for protecting software have been proposed so far, such as program obfuscation, program encryption, and tamper resistance techniques [3–5]. Many of the previous methods aim to protect against static analysis (i.e., analysis without running a program). In practice, however, the adversaries have the ability to handle dynamic analysis (i.e., analysis which is performed while running a program) as well as static analysis because they can easily obtain the powerful tools for dynamic analysis such as a debugger. Thus, there is a great demand for the method for protecting software against attacks based on both static and dynamic analysis [6].

This paper proposes a systematic software protection method which is resistant to not only static analysis, but dynamic analysis. We introduce the concept of *time-sensitive code*, which is dynamically modified at run-time according to the time (the number of clock cycles) taken to execute a block of the program. We focus on the point that the dynamic analysis with an intentional stop in a program (e.g., breakpoint, step-by-step execution) takes much more execution time than the normal execution does. If the execution time of the block is within a predetermined range, that is, the program is likely to be executed normally, the time-sensitive code becomes the original one. On the other hand, if the execution time is out of the range, we judge the dynamic analysis is being performed, and the time-sensitive code becomes the fake one. Since time-sensitive codes are scattered over the program, it is difficult for the adversaries to succeed in obtaining the secret information.

The time measurement technique itself is known for detecting if it runs under control of a debugger [5, 7]. When we simply apply this technique to increase the cost of obtaining secret information, it is easy for the adversary to find the secret information or to nullify the protection mechanism through static analysis since the secret information and the instructions for measuring time are not hidden. Then we combine the *instruction camouflage* method using self-modification mechanism [8, 9] with the time measurement technique in order to hide them and construct the program which is resistant to both static analysis and dynamic analysis.

The rest of this paper is organized as follows. Section 2 shows the adversary model we assume in this paper. Section 3 explains our method in detail. Section 4, we discuss the difficulty of attacking on a protected program. Section 5, we examine how much overhead on the execution time is imposed by the proposed method. In Section 6, we review the related work. Finally, Section 7 concludes the paper.

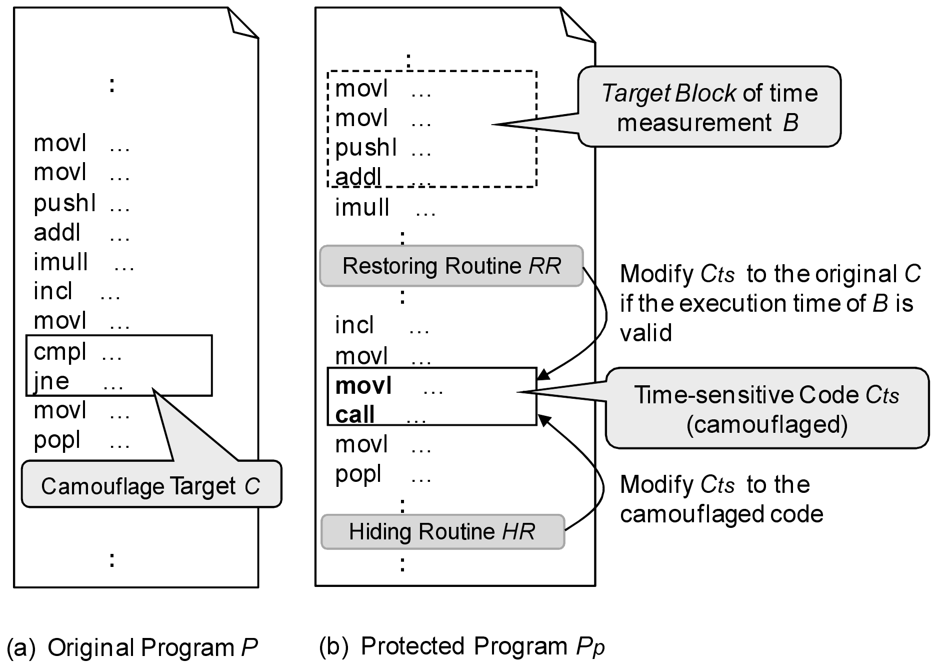


Figure 1. Basic idea of our method

2 Adversary Model

The *adversary* is assumed to be as follows.

- The adversary can perform static analysis. Specifically, the adversary has a tool for disassembling and the ability to inspect and modify all the instructions and data of the program in assembly language.
- The adversary can perform dynamic analysis. Specifically, the adversary has a debugging tool which enables the adversary to stop (and restart) the execution of the program and observe output of the program at an arbitrary point in the program. Using the tool, the adversary can also inspect and modify the state of the registers and the content of the program loaded in the memory at every program point.

The above adversary corresponds to the *level 2* adversary in the adversary model of Monden et al. [10]. This model also takes into account a *realistic model* described by Madou et al. [6]. Assuming the above adversary model, we propose and discuss our method in the following sections.

3 Protection Method

3.1 Basic Idea

First, we explain the basic idea of our method. Figure 1 (a) and (b) shows the basic concept of the original assembly program P and the protected assembly program P_p respectively. We show examples of assembly programs in the AT&T syntax based on the Intel x86 architecture [11] in this paper.

A part of the original program P is selected as the *camouflage target* C, which should be hidden from adversaries. C is overwritten with a camouflaged (fake) code. In the example of Figure 1, the instructions `cmpl` and `jne` are selected as the camouflage target, and the instructions are camouflaged as `movl` and `call` in P_p. The camouflaged code is called *time-sensitive code* C_{ts}. C_{ts} is modified during execution by *self-modification routines*, i.e., a *restoring routine* RR and a *hiding routine* HR, according to the time (the number of clock cycles) taken to execute the part of the program selected as B, the *target block* of time measurement. Specifically, RR rewrites C_{ts} to C if the execution time of B is within a predetermined range, that is, the program is likely to be executed normally. Otherwise, RR rewrites C_{ts} to the code that is different from C. If the adversary intentionally stops the program for analyzing purposes (e.g., examining the current state) at an instruction in B by means of a breakpoint or step-by-step execution, the execution time easily exceeds the predetermined range. As a result, C_{ts} is not modified to the original code C (C is not appeared in any registers and memory) in this case. In order to obtain C via dynamic analysis, the adversary has to stop the program at instructions between RR and HR without any stops at B. In order to increase the cost of obtaining C by the adversary, we build many time-sensitive codes in the program.

3.2 Procedure for constructing a protected program

A protected program P_p is obtained by repeating the following Step 1 to Step 6.

(Step 1) Determining the camouflage target C

First, the camouflage target code C is determined. A *user*, a person who uses our method, selects a part of P as C. C should be what the user wants to hide from adversaries. As an example of C, we take the following examples:

- The secret information itself such as constant values (secret keys) for DRM system, conditional branch instructions which branches based on input password, and main instructions of secret algorithm.

- Instructions or data which can be a clue to locate secret information such as an instruction for calling a dialog-box to input a password, and an (interrupt) instruction for operating an external device (e.g., a CD/DVD drive) to check if the product is legitimate or not.

- Instructions which contained in the protection mechanism such as restoring routines, hiding routines and instructions for measuring time. It is important for users to select these instructions as C and hide the existence of the protection mechanism in order to increase the cost of attack.

(Step 2) Generating camouflaged code

We generate C_{ts}, which aims to camouflage C. C_{ts} is determined at random, or the user may specify directly. The context of the program should be taken into account in order to make it difficult for adversaries to find C_{ts}. The generated C_{ts} overwrites C.

(Step 3) Determining B and the positions of the restoring/hiding routines

B and the positions of RR and HR in the program are determined. Below, the positions of RR and HR are denoted as $P(RR)$ and $P(HR)$, respectively.

First, B is determined at random from among the blocks composing P, or the user may specify B directly. A control flow graph (directed graph) with each instruction in the assembly program is considered as a node. $P(RR)$ and $P(HR)$ are chosen so that the following five conditions are satisfied. The conditions are intended to assure that C_{ts} is certain to be rewritten as C before it is executed, and is certain to be rewritten again as C_{ts} before the program ends, as long as the execution time of B is valid.

1. B is a basic block, that is, a linear sequence of program instructions having one entry point and one end point [12].

2. B must exist on every control flow path from the program entry to the $P(RR)$.

3. $P(RR)$ must exist on every control flow path from B to C_{ts}.

4. $P(HR_i)$ must not exist on every control flow path from $P(RR)$ to C_{ts}.

5. $P(RR)$ must exist on every control flow path from $P(HR)$ to C_{ts}.

6. $P(HR)$ must exist on every control flow path from C_{ts} to the end of the program.

(Step 4) Inserting instructions for measuring the time of B

The instructions for measuring the time of B are inserted to P. To measure the time of B, we use an instruction for counting clock cycles such as RDTSC (read time-stamp counter) instruction in Intel IA-32 architecture [13].

A simple example of measuring the time of a basic block using RDTSC instruction is shown below. The program includes the line numbers (in brackets) for ease of reference. In this example, the value which is in proportion to the number of clock cycles taken to execute the instructions between the RDTSC instructions (instructions at lines from 2 to 5) is obtained in the `edx` register.

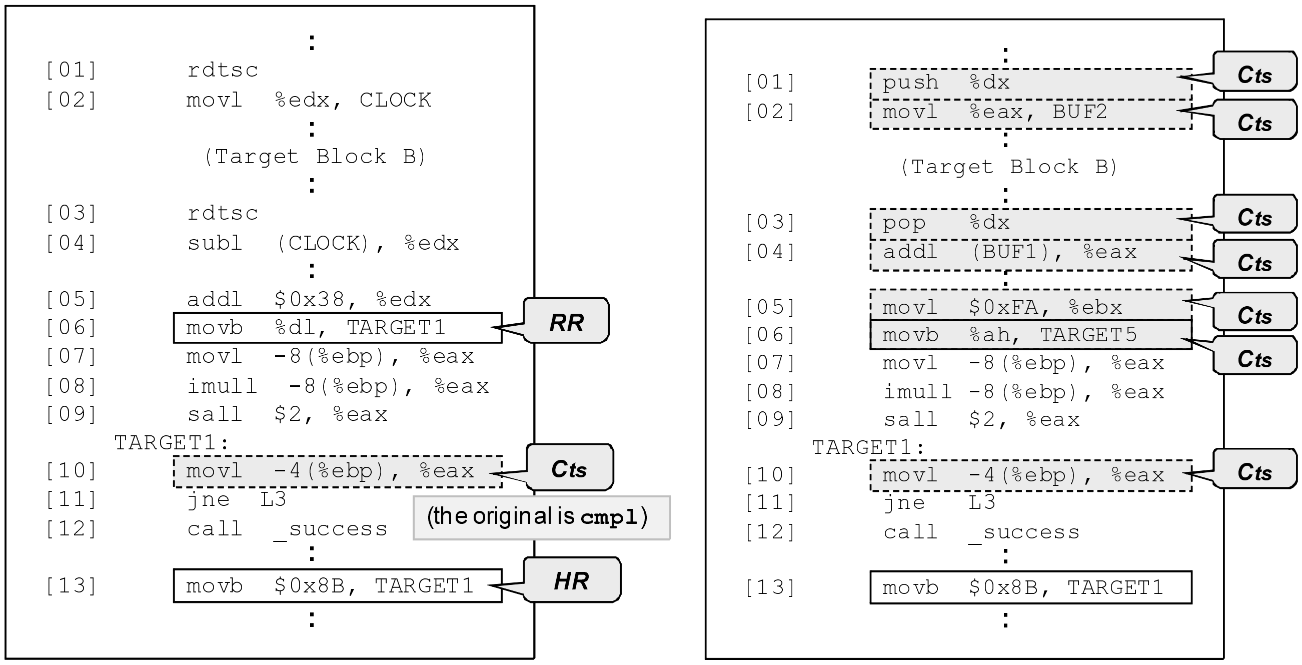
```
[01]  rdtsc
[02]  movl  %edx, CLOCK
[03]  movl  (%ebp), %ecx
[04]  imull (%ebp), %ecx
[05]  sall  $2, %ecx
[06]  rdtsc
[07]  subl  (CLOCK), %edx
```

(Step 5) Determining the threshold time

The threshold time to detect dynamic analysis (i.e., to judge whether the adversary performs dynamic analysis) is determined based on the content of B.

Now we define T_{min} as the minimum execution clock cycles, and T_{max} as the maximum execution clock cycles, and T_B as the number of clock cycles taken to execute B. We judge that the execution is normal if T_B is over T_{min} but less than T_{max}. Otherwise, we judge that dynamic analysis is performed or B is tampered with.

T_{min} and T_{max} is estimated based on the instructions composing B.



```
                    :
[01]      rdtsc
[02]      movl  %edx, CLOCK
                    :
           (Target Block B)
                    :
[03]      rdtsc
[04]      subl  (CLOCK), %edx
                    :
[05]      addl  $0x38, %edx
[06]      movb  %dl, TARGET1        RR
[07]      movl  -8(%ebp), %eax
[08]      imull  -8(%ebp), %eax
[09]      sall  $2, %eax
     TARGET1:
[10]      movl  -4(%ebp), %eax      Cts
[11]      jne  L3                   (the original is cmpl)
[12]      call  _success
                    :
[13]      movb  $0x8B, TARGET1      HR
                    :
```

```
                    :
[01]      push  %dx                 Cts
[02]      movl  %eax, BUF2          Cts
                    :
           (Target Block B)
                    :
[03]      pop   %dx                 Cts
[04]      addl  (BUF1), %eax        Cts
                    :
[05]      movl  $0xFA, %ebx         Cts
[06]      movb  %ah, TARGET5        Cts
[07]      movl  -8(%ebp), %eax
[08]      imull -8(%ebp), %eax
[09]      sall  $2, %eax
     TARGET1:
[10]      movl  -4(%ebp), %eax      Cts
[11]      jne   L3
[12]      call  _success
                    :
[13]      movb  $0x8B, TARGET1
                    :
```

(a) Program to which the procedure is applied once

(b) Program to which the procedure is applied many times

Figure 2. Example of a protected program

(Step 6) Generating restoring/hiding routines

Restoring routines RR and hiding routines HR are generated according to the following procedure:

1. A sequence of instructions which fulfills the following condition is constructed and is defined as RR: if $T_{min} \leq T_B < T_{max}$ is satisfied, then it modifies C_{ts} to C, otherwise, it modifies C_{ts} to a code which is different from C.

2. A sequence of instructions for modifying C to C_{ts} is constructed and is defined as HR.

Both restoring routines and hiding routines can be composed of instructions of high frequency such as `mov` instruction [8]. The generated RR and HR are inserted to the position which is determined in Step 3.

In order to make the self-modification routines difficult to find, the routines should be complicated by the use of conventional techniques such as obfuscation of machine language instructions [14] and mutation techniques [15].

3.3 Example

Figure 2 shows an example of an assembly program which is protected by the proposed method. This program has a simple password checking routine. The program includes the line numbers (in brackets) for ease of reference.

Figure 2(a) illustrates a part of the protected program to which the procedure shown in Section 3.2 was applied once. In Figure 2 (a), the instruction "`cmpl -4(%ebp), %eax`" is selected as C, and is overwritten by C_{ts} ("`movl -4(%ebp), %eax`"). For measuring T_B (the execution time of B), RDTSC instruction [13] is used. T_B is obtained by calculating the difference between the time-stamp counter (the system clock) just before starting B (which is obtained by the RDTSC instruction at line 1) and the one just after ending B (which is obtained by the RDTSC instruction at line 3). RR (at line 6) modifies C_{ts} according to T_B. The instruction `addl` (at line 5) calculates the value of the code which RR rewrites. In the normal execution, RR rewrites `movl` at line 10 to `cmpl`, that is, rewrites the first byte of `movl` to `3B`. The longer T_B is (i.e., the longer the adversary stops at B), the more the value of rewriting increases. If T_B is over T_{max}, RR rewrites C_{ts} to the code that is more than `3B`. Even if the adversary tampers with RR or removes instructions for measuring time, C_{ts} is not modified to C as long as the adversary cannot estimate the valid execution time correctly. HR (at line 13) rewrites the instruction which is rewritten by RR to the camouflage code `movl` again.

Figure 2(b) illustrates a part of the program which is obtained by applying the procedure to the program showed in Figure 2(a) many times. In Figure 2(b), the self-modification routines and the instructions for measuring time are selected as time-sensitive codes. As compared with Figure 2(a), it is difficult for the adversary to find that `movl` at line 10 is time-sensitive since the restoring routine and the instructions for measuring time are camouflaged. As seen in this example, we can increase the cost of attacking by repeating the procedure and constructing many time-sensitive codes.

4 Discussion

Here we discuss the difficulty of attacking on programs which is protected by the proposed method. Consider the case in which the adversary described in Section 2 attempts to obtain the secret part $S(P_p)$ in the protected program P_p. We assume that $S(P_p)$ is selected as C and is camouflaged as C_{ts}. The goal of the attack is defined as obtaining $S(P_p)$ correctly. Below we discuss the difficulty of achieving the goal by two types of attacks: static analysis and dynamic analysis.

4.1 Difficulty of Static Analysis

First, we discuss the case that the adversary performs static analysis to obtain $S(P_p)$.

In the protected program, $S(P_p)$ (i.e., the code which is selected as C) does not exist in the program since the code is camouflaged with C_{ts}. Thus, the adversary can not find $S(P_p)$ by searching for a specific instruction or data. For example, when the adversary searches for a camouflaged compare instruction in order to find a password checking routine, it is difficult to find the instruction because it does not exist in the program.

In addition, if instructions or data which can be a clue to locate secret information (e.g., an instruction for calling a dialog-box, an instruction for operating an external device) are camouflaged, it is difficult for the adversary to narrow the range of analysis to reduce the cost of program understanding.

In order to obtain the original content of the camouflaged code by means of static analysis, the adversary must find the restoring routines which are scattered over the program, and must guess the predetermined valid execution time of the target blocks (T_{min} and T_{max}), which requires a tremendous effort.

4.2 Difficulty of Dynamic Analysis

Next, we discuss the case that the adversary performs dynamic analysis to obtain $S(P_p)$. The adversary is able to run P_p using debugging tools, and try to identify and understand $S(P_p)$ based on the output information from the tools. The tools also enable the adversary to stop (and restart) the execution of the program and observe output of the program at an arbitrary point in the program.

When the adversary stops the execution at B to inspect or modify some code via debugging tools, T_B increases according to the stop time. If T_B exceeds T_{max}, the adversary can not obtain C because RR modifies C_{ts} to a code which is different from C. Even if the adversary tampers with RR or removes instructions for measuring time, C_{ts} is not modified to C as long as the adversary cannot estimate the valid execution time correctly.

In order to obtain C via dynamic analysis, the adversary must make the execution reach the instructions between RR and HR without stopping at B. The cost of obtaining C becomes more expensive if many time-sensitive codes are built in the program. (e.g., in case that the instructions between RR and HR is B for the other time-sensitive code).

5 Performance Overhead

In this section, we examine how much overhead on the execution time is imposed by the proposed method. The target is a program which decrypts 8 bytes of the encrypted data in the program, based on the 7 bytes of input data. The encryption algorithm used in the program is C2 (Cryptomeria Cipher) [16], which is designed for the CPPM(Content Protection for Prerecorded Media)/CPRM(Content Protection for Recordable Media) Digital Rights Management scheme.

First, we applied the proposed method to the subroutine of the program for decryption algorithm. Then, we measured the execution time of decrypting data 10,000 times for each version with different proportion of the time-sensitive instructions (C_{ts}) to the total instructions in the subroutine. The proportion of the time-sensitive instructions was varied from 0% to 30% with an interval of 10%.

The execution time was measured as the difference in the value of the processor's time-stamp counter (the number of clock cycles) using RDTSC instruction from just before the start of the protected program to just after the end of the program.

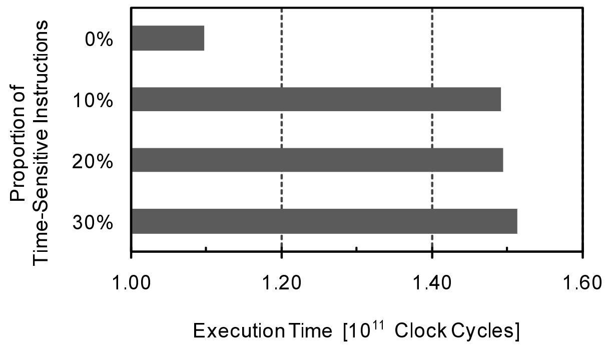


Figure 3. Overhead on the execution time

Figure 3 shows the result of the execution time measurement. The horizontal axis shows the average program execution time (clock cycles), while the vertical axis shows the proportion of time-sensitive instructions. It can be seen from Figure 3 that the average execution time tends to increase with the number of time-sensitive instructions. In particular, there is a large difference between the program that has no time-sensitive code (the original code) and the program which the proportion of time-sensitive instructions is 10%. When the proportion of time-sensitive instructions is approximately 30%, the average execution time is approximately 1.513 $\times 10^{11}$ clock cycles. This is approximately 1.38 times the execution time (approximately 1.097 $\times 10^{11}$ clock cycles) of the original program. We guess that the added instructions for constructing time-sensitive codes impose an extra overhead to the CPU. In particular, the self-modification mechanism can impose expensive overhead due to architectural issues such as incoherence of cache memory and prediction failure of conditional branches [17].

6 Related Work

Many methods for protecting secret information contained in software have been proposed so far, such as program obfuscation, program encryption and tamper resistance techniques [3–5].

The instruction camouflage [8, 9] is a straightforward obfuscation method using self-modification mechanism. In this method, some routines restore camouflaged instructions (i.e., instructions which are overwritten with dummy code) back to the original during execution. This method is especially effective in protecting against static analysis. There are some other protection methods using a self-modification mechanism, which includes the methods which dynamically decrypt the encrypted part of the program at some point in the execution [18, 19], and the method which mutates the program repeatedly during execution according to an *edit script* [20].

In addition, there are some anti-debugging methods using time measurement techniques such as [5, 7]. These methods use the techniques for detecting if it runs under control of a debugger. There is also a method for constructing a time-sensitive code exploiting characteristics of multiprocessor systems [21].

Our method aims to protect secret information against both static and dynamic analysis based on the combination of the instruction camouflage technique which is resistant to static analysis and the time measurement technique which helps detect dynamic analysis. Our method is systematic, thereby is easy to follow and easy to be automated.

7 Conclusion

In this paper, we proposed a systematic method for protecting software against malicious reverse engineering attacks, using time-sensitive codes based on self-modification mechanism.

In order to obtain the secret information by means of static analysis, the adversary must find the restoring routines within the whole program and must guess the target block and the predetermined valid execution time, which requires a tremendous effort. In order to obtain the secret information via dynamic analysis, the adversary must make the execution reach restricted points of the program without stopping the execution. The cost of obtaining the secret information becomes more expensive if many time-sensitive codes are built in the program.

It can be seen in the experiment about performance overhead that the more we build time-sensitive codes in the program, the more expensive the performance overhead becomes. That is, the more protected program suffers from the more overhead, which is a trade-off relation.

A systematic method for calculating the threshold time to detect dynamic analysis and its evaluation are left as a challenging issue in future work.

Acknowledgment

This research was supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 20700034.

References

[1] Y. Kanzaki, *Protecting Secret Information in Software Processes and Products* (PhD thesis, Nara Institute of Science and Technology, 2006).

[2] A. Monden and C. Thomborson, Recent software protection techniques – software-only tamper prevention –, *IPSJ Magazine*, 46(4), April 2005, 431–437 (In Japanese).

[3] C. Collberg and C. Thomborson, Watermarking, tamper-proofing, and obfuscation – Tools for software protection, *IEEE Transactions on Software Engineering*, 28(8), June 2002, 735–746.

[4] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Technical Report of Dept. of Computer Science, U. of Auckland, New Zealand, 1997.

[5] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection* (Addison-Wesley Professional, 2009).

[6] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, Hybrid static-dynamic attacks against software protection mechanisms, *Proc. The Fifth ACM Workshop on Digital Rights Management(DRM2005)*, Nov. 2005.

[7] N. Falliere, Windows anti-debug reference, http://www.symantec.com/connect/ja/articles/windows-anti-debug-reference , 2007.

[8] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, A software protection method based on instruction camouflage, *Wiley Publishers, Electronics and Communications in Japan, Part 3*, 89(1), January 2006, 47–59.

[9] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, Exploiting self-modification mechanism for program protection, *Proc. 27th IEEE Computer Software and Applications Conference*, Dallas, USA, Nov. 2003, 170–179.

[10] A. Monden, A. Monsifrot, and C. Thomborson, Tamper-resistant software system based on a finite state machine, *IEICE Transactions on Fundamentals*, E88-A(1), January 2005, 112–122.

[11] Intel Corporation, *IA-32 Intel Architecture software developer's manual vol.1 : Basic Architecture*, http://www.intel.co.jp/ (Available online).

[12] F. E. Allen and J. Cocke, A program data flow analysis procedure, *Communications of the ACM*, 19(3), 1976, 137–147.

[13] Intel Corporation, *IA-32 Intel Architecture software developer's manual vol.2 : Instruction Set Reference*, http://www.intel.co.jp/ (Available online).

[14] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software, *Proc. 1997 New Security Paradigm Workshop*, Sep. 1997, 23–33.

[15] J. Irwin, D. Page, and N.P. Smart, Instruction stream mutation for non-deterministic processors, *Proc. ASAP2002*, July 2002, 286–295.

[16] 4C-Entity. *Policy statement on use of content protection for recordable media, (CPRM) in certain applications*, 2001, http://www.4centity.com/ (Available online).

[17] Intel Corporation, *IA-32 Intel Architecture software developer's manual vol.3 : System Programming Guide*, http://www.intel.co.jp/ (Available online).

[18] D. W. Aucsmith, Tamper Resistant Software: An Implementation, *Lecture Notes in Computer Science*, vol. 1174, (Springer-Verlag, 1996), 317–333.

[19] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel, Self-encrypting code to protect against analysis and tampering, *Proc. Benelux Workshop on Information and System Security*, 2006.

[20] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, Software Protection through Dynamic Code Mutation, *Lecture Notes in Computer Science*, vol. 3786, (Springer-Verlag, 2006), 194–206.

[21] A. Torrubia and F. J. Mora, Information security in multiprocessor systems based on the x86 architecture, *Computers and Security*, 19(6), Oct. 2000, 559–563.