World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:7, 2008

Program Camouflage: A Systematic Instruction Hiding Method for Protecting Secrets

Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto

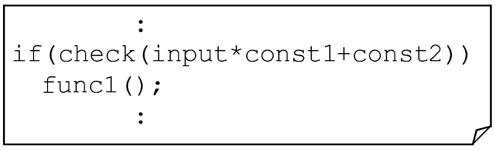
***Abstract*—This paper proposes an easy-to-use instruction hiding method to protect software from malicious reverse engineering attacks. Given a source program (original) to be protected, the proposed method (1) takes its modified version (fake) as an input, (2) differences in assembly code instructions between original and fake are analyzed, and, (3) self-modification routines are introduced so that fake instructions become correct (i.e., original instructions) before they are executed and that they go back to fake ones after they are executed. The proposed method can add a certain amount of security to a program since the fake instructions in the resultant program confuse attackers and it requires significant effort to discover and remove all the fake instructions and self-modification routines. Also, this method is easy to use (with little effort) because all a user (who uses the proposed method) has to do is to prepare a fake source code by modifying the original source code.

***Keywords*—Copyright protection, program encryption, program obfuscation, self-modification, software protection.

I. INTRODUCTION

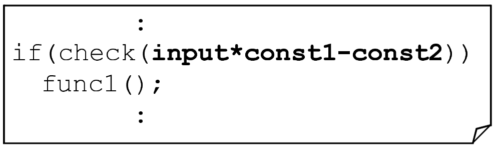
RECENTLY, many software products contain *secret information*, such as the cipher keys of a digital rights management system, conditional branch instructions for license checking, and algorithms that are commercially valuable [5], [12]. As the number such software increases, protecting internal secrets from being leaked out to software users via *reverse engineering* has become an overarching issue in today's software industry.

So far, various methods for protecting software against malicious reverse engineering attacks have been proposed, such as program encryption, program obfuscation and anti-debugging techniques [5]. All these methods can add a certain amount of security to software; however, they are often difficult to use especially when we consider hiding a specific piece of information in a program. For example, name obfuscation is one of the easy-to-use protection techniques that can hide all the symbols (variable names, function names and method names, etc.) in a program, however, it is unclear how it will contribute to conceal specific secrets, e.g. cipher keys and conditional branches in a program. On the other hand, more powerful protection techniques, such as control flow obfuscation, can directly hide the specific secret (conditional branches in this case), however, they are not always effective in obfuscating secret parts of a program that are required to be hidden.



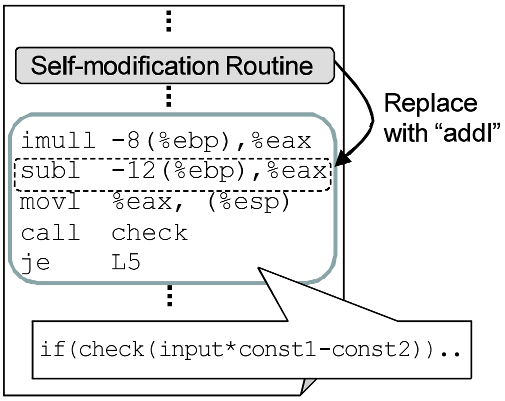
```
        :
if(check(input*const1+const2))
  func1();
        :
```

(a) Original Source Code



```
        :
if(check(input*const1-const2))
  func1();
        :
```

(b) Fake Source Code



```
imull -8(%ebp),%eax
subl  -12(%ebp),%eax
movl  %eax, (%esp)
call  check
je    L5
```

```
if(check(input*const1-const2))..
```

(c) Protected Code

Fig. 1 Concept of program camouflage

In this paper, we propose an easy-to-use, systematic method for protecting software which is able to hide an arbitrary part of a program with an arbitrary code. The proposed method enables us to determine how secret parts of a program are hidden at the source code level. All a user who uses the proposed method has to do is to construct a fake source code by modifying the original source code. When an attacker statically analyzes the (binary) program that is protected by the proposed method, the program appears the fake code. However, when the program is executed, the original code is performed. We call such a protection mechanism *program camouflage*.

Fig. 1 shows a basic concept of the program camouflage. Let us suppose that a source code to be protected has a secret conditional branch such as Fig. 1(a), and the user tries to fake the branch as Fig. 1(b) (the operator '+' is changed

Yuichiro Kanzaki is with the Department of Information and Computer Sciences, Kumamoto National College of Technology, Koshi, Kumamoto, Japan, email: kanzaki@knct.ac.jp

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

Masahide Nakamura is with the Graduate School of Engineering, Kobe University, Kobe, Hyogo, Japan, email: masa-n@cs.kobe-u.ac.jp

Ken-ichi Matsumoto is with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, Nara, Japan, email: matumoto@is.naist.jp

Open Science Index, Computer and Information Engineering Vol:2, No:7, 2008 publications.waset.org/12648/pdf

to '-'). The proposed method first analyzes the differences in assembly code instructions between original and fake, and then self-modification routines are introduced so that fake instructions become correct (i.e., original instructions) before they are executed. In this example, the self-modification routine rewrites the instruction "`subl -12(%ebp),%eax`" (subtraction) to the instruction "`addl -12(%ebp),%eax`" (addition), when the program is executed (Fig. 1(c)) [1] .

In this manner, a user can hide an arbitrary part of the program with an arbitrary code simply by preparing a fake source code. In addition, the user is not required to possess knowledge about assembly (machine) language. We believe that these advantages lead to efficient program protection.

The rest of this paper is organized as follows. In Section II, we review the related work. Section III describes the proposed method in detail. In Section IV, we show an example of the camouflaged program and discuss it. In Section V, we examine how much overhead on the execution time is imposed by the proposed method. Finally, Section VI concludes the paper.

II. RELATED WORK

The basic idea of the proposed method is based on *instruction camouflage* method, which is previously proposed by the authors [13], [14]. Instruction camouflage is done according to the following procedure. First, many instructions are randomly selected as the targets of instruction camouflage at the assembly language level. Second, fake instructions to overwrite the targets are generated. The content of fake instructions is determined at random. Finally, self-modification routines are generated and inserted into the program. This procedure is very simple and easy to automate camouflaging programs. However, instructions that should be hidden from attackers (i.e., instructions that can be a clue to success in analyzing the program) may be left un-camouflaged. Besides, in order to hide the specific instructions manually, the user is required to be familiar with assembly language. With the proposed method, on the other hand, the user is able to hide instruction that the user wants to hide without knowledge about assembly language since the user can determine the content of the camouflage at the source code level.

In addition, many methods for protecting software against malicious reverse engineering attacks have been proposed so far. Program encryption [2], [4], [6] is a method for protecting programs using an encryption algorithm. A part of the program is encrypted beforehand and is decrypted by self-modification during execution. Program obfuscation [5], [7], [17] is a technique in which a program is modified in a way such that it becomes more difficult to analyze (more complex) it without modifying its specifications. Anti-debugging [3] is a technique that aims to deter attackers from attacking with a debugger. The proposed method is a technique that is not controversial with respect to program encryption, program obfuscation, or anti-debugging. Consequently, analysis of the program is made even more difficult by combining the proposed method with these approaches.

III. PROPOSED METHOD

A. Outline of program camouflage

We first outline program camouflage method and provide some definitions. Fig. 2 shows an outline of program camouflage.

The *original source code* S is a source code to be protected. A user who uses the proposed method (e.g., a program developer) prepares a *fake source code* S_f by modifying S (Step 1). The assembly codes obtained by compiling S and S_f are denoted as *original assembly code* A and *fake assembly code* A_f, respectively.

We then compare A and A_f and determine what operations are needed to obtain A from A_f (Step 2). Such an operation is defined as a *restoring operation*, which has three types: *change*, *add*, and *delete*. $RO(A_f, A)$ is a set of restoring operations that are needed to obtain A from A_f.

Next, self-modification routines based on the restoring operations are generated (Step 3). The self-modification routine has two types: restoring routine and hiding routine. A restoring routine restores the original instruction hidden by a fake instruction. In contrast, a hiding routine rewrites a restored instruction to a fake instruction.

Finally, by adding the self-modification routines to A_f, the user can obtain a *camouflaged assembly code* A_c that is functionally equivalent to the original one, but is more complex for attackers to analyze (Step 4).

The details of each step are described in the following section.

B. Procedure of program camouflage

(Step 1) Constructing a fake source code

S_f is constructed by changing, adding, and/or deleting instructions in S. In the example illustrated in Fig. 2(a) and (b), the instruction
"`if(key1*10 +key2==45) break;`"
in S is changed to
"`if(key1*5<=25) break;`"
in S_f.

The content of S is not restricted as long as both S and S_f are compilable by the same compiler. However, the larger the difference between S and S_f, greater is the number of self-modification routines that are required, which causes an increase in the performance overheads. The performance overhead imposed by the proposed method will be described in Section V.

(Step 2) Differential analysis

We compile S and S_f and get their assembly codes A and A_f, respectively. We then compare A and A_f and get the restoring operations $RO(A_f, A)$. Specifically, we use Myers's difference algorithm [16] to obtain $RO(A_f, A)$.

The restoring operation has three types as follows:

$change(i_f, i)$: The operation of changing the instruction i_f in A_f to the instruction i in A.

$add(i_f, i)$: The operation of adding the instruction i in A just after the instruction i_f in A_f.

---

[1] In this paper, it is assumed that the type of CPU used is Intel X86 and show assembly codes in AT&T syntax.

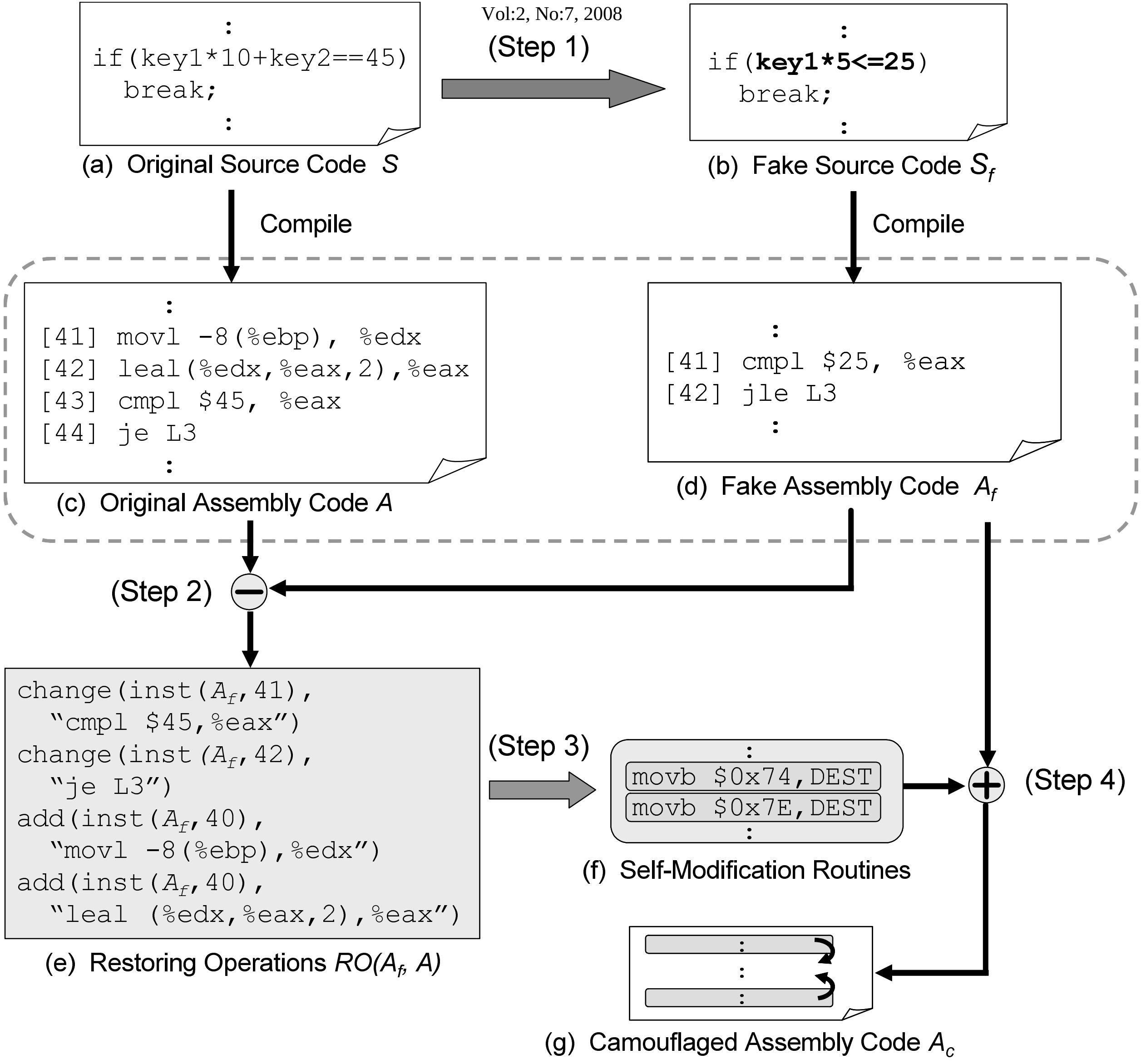


Fig. 2 Outline of program camouflage

$delete(i_f)$: The operation of deleting the instruction i_f in A_f.

If all restoring operations in $RO(A_f, A)$ are done with A_f, A_f becomes a program functionally equivalent to A.

Fig. 2(c) and (d) shows an example of A and A_f, which correspond to Fig. 2(a) and (b), respectively. The numbers in brackets in Fig. 2(c) and (d) indicate the serial numbers of the instructions in the program.

A_f does not have the following instructions:

- "`movl -8(%ebp),%edx`" (the 41st instruction in A)
- "`leal (%edx,%eax,2),%eax`" (the 42nd instruction in A).

This is because the instruction "`key1*10+key2`" in S has been changed to "`key1*5`" in S_f. Similarly, A_f has the instructions "`cmpl 25,%eax`" and "`jle L3`" instead of "`cmpl 45,%eax`" and "`je L3`", since "`==45`" in S has been changed to "`<=25`" in S_f.

In this example, $RO(A_f, A)$ consists of four restoring operations as follows:

- $change(inst(A_f, 41)$, "`cmpl $45, %eax`")
- $change(inst(A_f, 42)$, "`je L3`")
- $add(inst(A_f, 40)$,
  "`movl -8(%ebp), %edx`")
- $add(inst(A_f, 40)$,
  "`leal (%edx,%eax,2),%eax`")

where $inst(A_f, j)$ means the j-th instruction in A_f.

(Step 3) Generating self-modification routines

Self-modification routines are generated based on $RO(A_f, A)$. Now, RR_k and HR_k are defined as the restoring routine and the hiding routine, respectively, for the k-th restoring operation d_k in $RO(A_f, A)$, and $dest$ is defined as the address of the instruction (i.e., location of the instruction) modified by RR_k or HR_k. Also, src_R (or src_H)

is defined as the instruction that overwrites the instruction at $dest$ by RR_k (or HR_k).

RR_k and HR_k are generated as below.

(1) First, src_R, src_H, and $dest$, are determined according to the type of d_k.
(1-a) If d_k is $change(i_f, i)$, src_R is identical to i, and src_H is identical to i_f. $dest$ is the address of i_f.
(1-b) If d_k is $add(i_f, i)$, an instruction i_a, whose length is the same as that of i, is first generated. Then, i_a is inserted just after i_f in A_f. src_R is identical to i, and src_H is identical to i_a. $dest$ is the address of i_a.
(1-c) If d_k is $delete(i_f)$, an instruction i_n is first generated in a random manner, which does not change the execution state (i.e., does not change the values of the registers) when it is executed at the address of i_f. src_R is identical to i_n, and src_H is identical to i_f. $dest$ is the address of i_f.
(2) We compare src_R with src_H at the binary code level, and construct instruction(s) to write a number of bytes, to turn src_H into src_R. This instruction(s) is defined as RR_k.
(3) In the same way, we construct instruction(s) to write a number of bytes, to turn src_R into src_H. This instruction(s) is defined as HR_k.

According to the procedure described above, we provide an example of how to generate self-modification routines. In this example, we generate RR_k and HR_k for the restoring operation $change$ $(inst(A_f, 42)$, "`je L3`") shown in Fig. 2(e).

(1) Since d_k is $change(inst(A_f, 42)$,"`je L3`"), src_R is "`je L3`", and src_H is the 42nd instruction in A_f, that is, "`jle L3`". $dest$ is the address of the 42nd instruction in A_f.

(2) It is assumed that the binary representation (in hex) of src_R("`je L3`") and src_H("`jle L3`") is "`74 11`" and "`7E 11`", respectively.
In order to turn src_H at $dest$ into src_R, RR_k changes the first byte of the instruction at $dest$ from "`7E`" to "`74`". When a label `DEST` points to $dest$, RR_k can be generated as follows:

```
movb $0x74,DEST
```

This routine means that the first byte of the instruction where the label `DEST` is pointing is overwritten with the immediate value "`74`" in hex. When this routine runs, the instruction at $dest$ is set to src_R.

(3) In the same way as (2), HR_k is generated.
In order to turn src_R at $dest$ into src_H, HR_k changes the first byte of the instruction at $dest$ from "`74`" to "`7E`".
When a label `DEST` points at $dest$, RR_k can be generated as below:

```
movb $0x7E,DEST
```

When this routine runs, the instruction at $dest$ is set to src_H.

(Step 4) Inserting self-modification routines

Finally, we insert the self-modification routines to A_f and obtain A_c.

We can automatically determine the position of inserting the self-modification routines based on the instruction camouflage method [13], [14].

IV. EXAMPLE OF PROGRAM CAMOUFLAGE

In this section, we provide an example of camouflaging a program. It is assumed that the user has constructed the fake source code S_f shown in Fig. 3(b) from the original source code S shown in Fig. 3(a). Specifically, the following operations have been done with S:

- The instruction "`if(key1 * 10 + key2 == 45) break;`" has been deleted.
- The instruction "`if(key1 + key2 <= 70)`" has been added and connected with the instruction "`printf("Password OK.\n")`".
- The instruction "`if(key1 * 5 <= 25)`" has been added and connected with the instruction "`printf("Invalid Password.\n")`".
- The `while` statement has been deleted (the control structure of the program has been changed).

In this example, $RO(A_f, A)$ consists of nine restoring operations as follows (refer to the numbers in brackets in Fig. 3(c) for the serial numbers of the instructions in A_f):

- $change(inst(A_f, 14)$, "`imull $10,-4(%ebp),%eax`")
- $change(inst(A_f, 16)$, "`addl -8(%ebp),%eax`")
- $change(inst(A_f, 17)$, "`cmpl $45,%eax`")
- $change(inst(A_f, 18)$, "`je _L2`")
- $change(inst(A_f, 22)$, "`jmp L2`")
- $delete(inst(A_f, 23))$
- $delete(inst(A_f, 24))$
- $delete(inst(A_f, 25))$
- $delete(inst(A_f, 28))$

Fig. 3(c) shows an example of the camouflaged assembly code. In Fig. 3(c), the labels "`DEST1:`", "`DEST2:`", ..., "`DEST`k`:`" ..., "`DEST9:`" point to addresses to be modified by self-modification routines RR_k and HR_k.

The camouflaged assembly code consists of instructions only included in the fake assembly code and self-modification routines. Therefore, an attacker will not be able to understand the program as long as the attacker reads only a part of it. For example, if the attacker reads only the part consisting of the following instructions:

```
[16] addl -4(%ebp),%eax
[17] cmpl $70,%eax
[18] jg _L2 ,
```

it will seem as if the operation "`key1+key2<=70`" was performed. In fact, this part is modified by restoring routines at run-time, and the operation "`key1*10+key2==45`" is performed.

For simplification, we have shown one of the smallest self-modification routines in this example. In order to make static

```
int main() {
  int key1, key2;

  while(1) {
    scanf("%d", &key1);
    scanf("%d", &key2);

    if(key1 * 10 + key2 == 45)
      break;

    printf("Invalid Password.¥n");
  }
  printf("Password OK.¥n");
  return 0;
}
```

(a) Original Source Code

```
int main() {
  int key1, key2;

  scanf("%d", &key1);
  scanf("%d", &key2);

  if(key1 + key2 <= 70)
    printf("Password OK.¥n");

  if(key1 * 5 <= 25) {
    printf("Invalid Password.¥n");
  }

  return 0;
}
```

(b) Fake Source Code

```
 Constant Declaration
LC1:.ascii "%d¥0"
LC3:.ascii "Invalid Password.¥0"
LC4:.ascii "Password OK.¥0"
 Pre-processing
_main:
      movb  $0xF8, DEST2+2     ··· RR2
[01]  pushl %ebp
[02]  movl  %esp, %ebp
[03]  pushl %edx
[04]  pushl %edx
[05]  call  ___main
      subb  $0x07, DEST5+1     ··· RR5
L2:
      movb  $0x6B, DEST1        }
      movw  $0x0AFC, DEST1+2    } RR1
 Get "key1" and "key2"
      movw  $0x9090, DEST7      }
      movb  $0x90, DEST7+2      } RR7
[06]  leal  -4(%ebp), %eax
[07]  pushl  %eax
[08]  pushl  %eax
      movw  $0x9090, DEST8     ··· RR8
[09]  pushl $LC1
[10]  call  _scanf
[11]  leal  -8(%ebp), %eax
[12]  pushl %eax
[13]  pushl %eax
      movb  $0x74, DEST4       ··· RR4
[14]  pushl $LC1
[15]  call  _scanf
      movb  $0x2D, DEST3+2     ··· RR3
 Fake as "if(key1+key2 <= 70)"
DEST1:  # change to
        # "imull $10,-4(%ebp),%eax"
[16]  movl  -8(%ebp), %eax
      ret   # added (padding)
[17]  addl  $24, %esp
      movb  $0x8B, DEST1        }
      movw  $0xF845, DEST1+2    } HR1
      movb  $0x90, DEST9       ··· RR9
```

```
DEST2:  # change to
        # "addl -8(%ebp),%eax"
[18]  addl  -4(%ebp), %eax
DEST3:  # change to "cmpl $45,%eax"
[19]  cmpl  $70, %eax
DEST4:  # change to "je _L2"
[20]  jg    _L2
[21]  pushl $LC3
[22]  call  _puts
[23]  popl  %edx
      movl  $0x90909090, DEST6 ··· RR6
DEST5:  # change to "jmp L2"
[24]  jmp  L2+7
 Fake as "if(key1*5 <= 25)"
_L2:
      movb  $0x45, DEST3+2       ··· HR3
DEST6:  # delete
[25]  imull $5, -4(%ebp), %eax
DEST7:  # delete
[26]  cmpl  $25, %eax
DEST8:  # delete
[27]  jg    _L3
[28]  pushl  $LC4
      movw  $0x7F27, DEST8       ··· HR8
[29]  call  _puts
DEST9:  # delete
[30]  popl  %eax
_L3:
      movw  $0x83F8, DEST7        }
      movb  $0x19, DEST7+2        } HR7
      movl  $0x6B45Fc05, DEST6 ··· HR6
      addb  $0x07, DEST5+1       ··· HR5
      movb  $0x7F, DEST4         ··· HR4
 Post-processing
[31]  leave
[32]  xorl  %eax, %eax
      movb  $0xFC, DEST2+2       ··· HR2
      movb  $0x58, DEST9         ··· HR9
[33]  ret
```

(c) Camouflaged Assembly Code

Fig. 3 Example of program camouflage

analysis difficult, self-modification routines can be made complex using other obfuscation methods, such as the obfuscation of the assembly code [15] and mutation [11]. In addition, the distance between fake instructions and the self-modification routines tends to be small, since this example program is very short. However, if the size of the target program is large, self-modification routines can be scattered over a wide range, which further increases the cost of analyzing the program.

V. PERFORMANCE OVERHEAD

In this section, we examine how much overhead on the execution time is imposed by the proposed method. The target of camouflage 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) [1], which is designed for the CPPM(Content Protection for Prerecorded Media)/CPRM(Content Protection for Recordable Media) Digital Rights Management scheme.

First, we camouflaged the subroutine of the program for decryption algorithm. Then, we measured the execution time 10 times for each version with different *proportion of the camouflaged instructions* to the total instructions in the subroutine. By the proportion of camouflaged instructions, we characterize the degree of the camouflage in the program, which is in proportion to the degree of the difference between the original assembly code and the fake assembly code. The proportion of the camouflaged instructions was varied from 10% to 50% with an interval of 10%.

The content of fake source code here was determined at random for simplicity. The computer used in the experiment had Windows XP as the OS, a main memory size of 1.5 Gbytes, and a Pentium 4 CPU (clock frequency 2.8 GHz) based on IA-32 (Intel Architecture 32) [8]. The execution time was measured as the difference in the value of the processor's time-stamp counter from immediately before the start of the camouflaged program to immediately after the termination of the program. The value of the processor's time-stamp counter was acquired by using the `RDTSC` (Read Time-Stamp Counter) instruction [9].

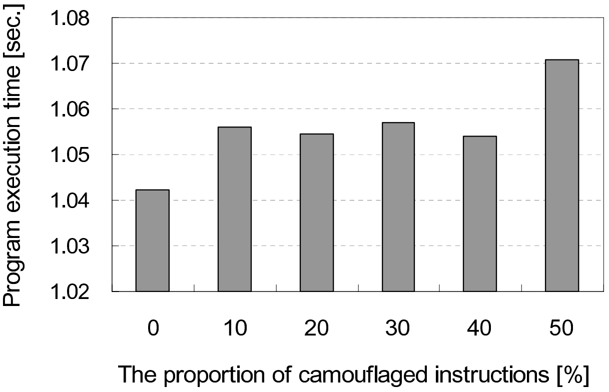


Fig. 4 Overhead on the execution time

Fig. 4 shows the result of the execution time measurement. The horizontal axis shows the proportion of camouflaged instructions, while the vertical axis shows the average program execution time. It can be seen from Fig. 4 that the average execution time tends to increase with the number of camouflaged instructions. When approximately 50% of the entire instructions are camouflaged, the average execution time is approximately 1.07 seconds. This is approximately 1.03 times the execution time (approximately 1.04 second) when no instruction is camouflaged. We guess that the self-modification mechanism imposes an extra overhead to CPU, due to architectural issues such as incoherence of cache memory, or prediction failure of conditional branches [10].

VI. CONCLUSION

In this paper, we have proposed program camouflage method. All a user who uses the proposed method has to do is to construct a fake source code by modifying the original source code. When an attacker statically analyzes the program that is protected by the proposed method, the program appears the fake code. However, when the program is executed, the original code is performed.

Since a user can determine the content of the camouflage at the source code level, it is easy to hide specific instructions or constant values of the program from attackers. For example, consider that (part) instruction "`(a+b)*5==10`" is to be converted into fake instruction "`a*8>=10`". For this conversion by the (previous) instruction camouflage method, the user has to read the assembly code and understand the behavior of the program (e.g., the control flow, data flow, and stack operations). On the other hand, with the proposed method, the user simply has to change the "`(a+b)*5==10`" to "`a*8>=10`" at the source code level. In addition, it is easy to camouflage a program on a module basis, not on an instruction basis. For example, a user can fake an encryption algorithm E_1 used in the program as another encryption algorithm E_2, simply by replacing E_1 with E_2 at the source code level. We believe that these advantages lead to efficient program camouflage.

It can be seen in the experiment that the more the proportion of camouflaged instructions increases, the more expensive the performance overhead becomes. Therefore, too much camouflage should not be applied to such programs that require high performance or real-time properties. On the other hand, programs that can sacrifice performance but requires a strong protection have a benefit of the high degree of camouflage. Thus, the proposed method should be applied with a careful consideration on the target program itself and the objective of the protection.

ACKNOWLEDGMENT

The research was supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 20700034, 2008(Japan).

REFERENCES

[1] 4C-Entity, *Policy statement on use of content protection for recordable media, (CPRM) in certain applications*, 2001, http://www.4centity.com/ (Available online).

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

[3] P. Cervan, *Crackproof Your Software*. San Francisco: No Starch Press, 2002.

[4] F. Cohen, Operating system protection through program evolution, *Computers and Security*, vol. 12, no. 6, pp. 565–584, 1993.

[5] C. Collberg and C. Thomborson, Watermarking, tamper-proofing, and obfuscation – tools for software protection, *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, June 2002.

[6] D. Grover, Ed., *The Protection of Computer Software: Its Technology and Applications*. Cambridge University Press, 1989.

[7] F. Hohl, *Time limited blackbox security: Protecting mobile agents from malicious hosts*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1998, vol. 1419, pp. 92–113.

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

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

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

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

[12] Y. Kanzaki, Protecting secret information in software processes and products, Ph.D. dissertation, Nara Institute of Science and Technology, Mar. 2006.

[13] 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*, vol. 89, no. 1, pp. 47–59, January 2006.

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

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

[16] E. W. Myers, An O(ND) difference algorithm and its variations, *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[17] C. Wang, J. Hill, J. Knight, and J. Davidson, Software tamper resistance: Obfuscating static analysis of programs, Department of Computer Science, University of Virginia, Technical Report SC-2000-12, Dec. 2000.