Electronics and Communications in Japan, Part 3, Vol. 89, No. 1, 2006
Translated from Denshi Joho Tsushin Gakkai Ronbunshi, Vol. J87-A, No. 6, June 2004, pp. 755–767

A Software Protection Method Based on Instruction Camouflage

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

Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma, 630-0192 Japan

SUMMARY

This paper presents a method in which program analysis by a malicious user (attacker) is made difficult by camouflaging (hiding) a large number of instructions contained in the program. In the proposed method, an arbitrary instruction (target) in the program is camouflaged by a different instruction. Using the self-modification mechanism in the program, the original instruction is restored only in a certain period during execution. Even if the attacker attempts an analysis of the range containing the camouflaged instruction, it is impossible for him to correctly understand the original behavior of the program unless he notices the existence of the routine that rewrites the target (restoring routine). In order to make the analysis a success, the range containing the restoring routine must be analyzed, and the attacker is forced to analyze a wider range of the program. The proposed method can easily be automated, and the number of targets can be specified arbitrarily according to the required degree of protection and the acceptable degradation of execution efficiency. © 2005 Wiley Periodicals, Inc. Electron Comm Jpn Pt 3, 89(1): 47–59, 2006; Published online in Wiley InterScience (www.interscience.wiley.com). DOI 10.1002/ecjc.20141

Key words: copyright protection; software protection; program obfuscation; program encryption; self-modification.

1. Introduction

With the spreading use of networks, there has been remarkable progress in the flow configuration of programs and digital content. Accompanying this, there is an increasing demand for techniques which can prevent internal analysis and tampering with programs by end users. In programs containing digital rights management (DRM), for example, it is necessary to prevent interception of the internal decryption key [3, 24]. In a program built into the hardware of portable phones and set-top boxes, it is also required to prevent analysis or tampering by the user [23]. An example of the problems caused by analysis is the phenomenon in which a decryption tool for DVD data was disseminated [6, 22]. This tool was based on an analysis of a DVD playback program, and greatly facilitated illegal copying of DVDs.

By analysis in this paper is meant an attempt to acquire secret information (such as a secret key or algorithm) in a program. Typically, such an action is assumed to involve the following steps. First the attacker disassembles the program and tries to understand the resulting assembly program [17]. However, a tremendous amount of labor and time is required to understand the entirety of a large-scale program, which is not realistic. Consequently, the attacker restricts the range to be considered (the range which seems to be related to the secret information), and tries to understand only that range [1, 2]. The restriction and understanding of the range is repeated until the desired secret information is acquired.

This paper proposes a method in which a large number of instructions in the program are camouflaged (hidden) in order to make it difficult to analyze an assembly language program accompanied by such range restriction. In the proposed method, an arbitrary instruction (target) in the program is camouflaged by a different instruction. By using a self-modification mechanism in the program, the original instruction is restored only in a certain period during execution [15]. Even if the attacker attempts an analysis of the range containing the camouflaged instruction, it is impossible for him to correctly understand the original behavior of the program unless he notices the existence of the routine that rewrites the target (restoring routine). In order make the analysis a success, the range containing the restoring routine must be analyzed, and the attacker is forced to analyze a wider range of the program. The proposed method can easily be automated, and the number of targets can be specified arbitrarily. By distributing a large number of targets and a large number of restoring routines in the program, it is likely that analysis by range restriction will be made very difficult.

Below, Section 2 proposes a systematic method which camouflages a large number of instructions in the program by self-modification. Section 3 discusses attacks on the proposed method, and analyzes the difficulty of the attack and its prevention. Section 4 reports a case study using the proposed method. Section 5 describes related studies. Section 6 gives conclusions and describes the problems remaining.

2. Method of Software Protection by Instruction Camouflage

2.1. Attacker model

In this paper, the attacker model is assumed to be as follows.

- The attacker has a disassembler and the ability to perform static analysis including range restriction by using the disassembler.
- The attacker has a debugger with a break-point function. By (manually) setting the break-point at an arbitrary point in the program, a snapshot at an arbitrary execution time (i.e., the content of the program which is the object of analysis loaded in the memory) can be acquired. However, he does not have tools by which a snapshot can be acquired automatically or by which dynamic analysis using the acquired snapshot history can be automated. He also lacks the ability to construct such a tool.

The above attacker corresponds to the "level 2 attacker" in the graded attacker model of Monden and colleagues [18].

Assuming the above attacker model, the mechanism of program protection must satisfy the following requirements.

- The protection mechanism is not easily invalidated by static analysis using a disassembler.
- The protection mechanism is not easily invalidated by an attack using (a few) snapshots.

In the following sections, a protection method satisfying the above properties is proposed.

2.2. Central idea

The proposed method makes it difficult for the attacker to understand the program by camouflaging program instructions. By camouflage is meant hiding the existence of the original instructions from the attacker by overwriting the instructions by false instructions with different content.

Figure 1 shows an example of camouflage.[*] Consider the situation in which the instruction `jne L10` in the assembly program to be protected is camouflaged. First, a dummy instruction for `jne L10`, that is, an instruction with different content, is constructed. Suppose that `jmp L7` is constructed as the dummy instruction. Then, overwriting with `jmp L7` is performed at the position of `jne L10`.

Then a self-modification routine is added. By self-modification is meant the process of modifying the content of instructions in the program during execution. There are two kinds of self-modification routines. One is a routine that rewrites the camouflaged instruction to the original content (*RR* in Fig. 1). This routine assures the execution of the original content of the program. In the case of Fig. 1, it is the routine to rewrite the camouflaged instruction as `jne L10`.

The other is a routine that again rewrites an instruction which has been returned to the original instruction by *RR* as the dummy instruction (*HR* in Fig. 1). This routine is intended to hinder an attacker who has a snapshot acquisition capability from determining the original instruction. In the case of Fig. 1, it is a routine that automatically rewrites the instruction to be camouflaged as `jmp L7`.

The dummy instruction takes the form of the original instruction only between the execution of *RR* and the execution of *HR*. Consequently, it is difficult for the attacker to determine that the original instruction is overwritten by the dummy instruction `jmp L7` by simply observing the neighborhood of the camouflaged instruction. Even if the snapshot of the program after execution of *HR* is acquired, it is impossible to determine the original instruction from the obtained snapshot.

The above instruction camouflage is repeated several times on the assembly program to be protected, in order to make the program difficult to understand. Figure 2 shows conceptually the program obtained after instruction camou-

[*]This paper assumes an Intel ×86 type CPU as an example for description. The assembly language instructions are based on the AT&T syntax.

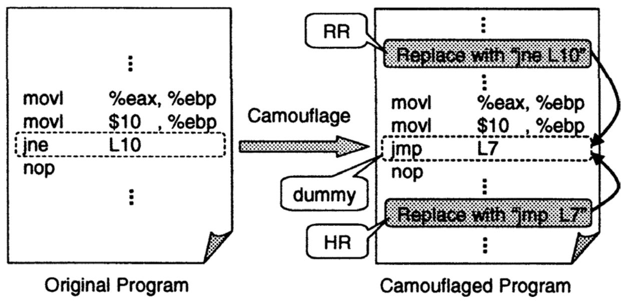


Fig. 1. Example of camouflage.

flaging has been repeated many times. It is evident that a large number of instructions in the program have been overwritten by dummy instructions before execution (● in Fig. 2). For each dummy, there exists a routine that rewrites the instruction to the original instruction (before it was overwritten by the dummy instruction) (■ in Fig. 2), and the routine that during execution takes the instruction rewritten to the original instruction by the above routine and rewrites it again as the dummy instruction (▲ in Fig. 2).

If the part of the program which the attacker attempts to analyze contains a dummy instruction, he cannot correctly determine the original behavior of the program by examining only that part. In order to understand the program correctly, he must know that rewriting is performed, and determine the content of each dummy instruction in the part to be understood before it was overwritten. In order to obtain this information, however, he must locate the routine that rewrites the instruction to the original instruction within the whole program, which requires a tremendous effort.

2.3. Definitions

The terminology in the proposed method is defined as follows. The original program O is the program before camouflaging is to be applied. The target instruction is the

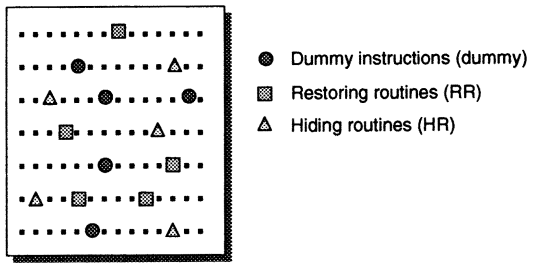


Fig. 2. Image of a camouflaged program.

instruction which is the target of camouflaging in O. A dummy instruction is an instruction which is written over the target instruction in order to camouflage the target instruction. When the user defines multiple target instructions, the i-th target instruction is written as $target_i$ and the instruction used to camouflage $target_i$ is written as $dummy_i$.

The restoring routine is a routine (a series of instructions) that rewrites an instruction camouflaged by the dummy instruction, thus restoring the original target instruction. The restoring routine that rewrites $dummy_i$ to $target_i$ is denoted as RR_i. The hiding routine, on the other hand, is a routine that rewrites the target instruction to the dummy instruction. The hiding routine that rewrites $target_i$ to $dummy_i$ is denoted as HR_i. The restoring routine and the hiding routine are together called self-modification routines.

A camouflaged instruction is an instruction whose content is changed (to $target_i$ or $dummy_i$) during execution. A camouflaged program M is an assembly program which contains camouflaged instructions. In the following sections, a systematic method is presented for deriving the camouflaged program M from the original program O.

2.4. Construction of camouflaged program *M*

M is constructed by the following steps 1 to 6.

(Step 1) Determination of the target instruction and the position of the self-modification routines

$target_i$ and the positions of RR_i and HR_i in the program are determined. Below, the positions of RR_i and HR_i are denoted as $P(RR_i)$ and $P(HR_i)$, respectively. First, $target_i$ is determined at random from among the instructions composing M. Or, the program developer may specify the target instruction directly. We consider a control flow graph (directed graph) with each instruction in the assembly program as a node. $P(RR_i)$ and $P(HR_i)$ are chosen so that the following four conditions are satisfied. The conditions are intended to assure that $dummy_i$ is certain to be rewritten as $target_i$ before it is executed, and is certain to be rewritten again as $dummy_i$ before the program ends.

[Condition 1] $P(RR_i)$ exists on any path from *start* to $target_i$.

[Condition 2] $P(HR_i)$ does not exist on any path from $P(RR_i)$ to $target_i$.

[Condition 3] $P(RR_i)$ exists on any path from $P(HR_i)$ to $target_i$.

[Condition 4] $P(HR_i)$ exists on any path from $target_i$ to the end of the program.

Figure 3 shows an example of $P(RR_i)$ and $P(HR_i)$ satisfying conditions 1 to 4.

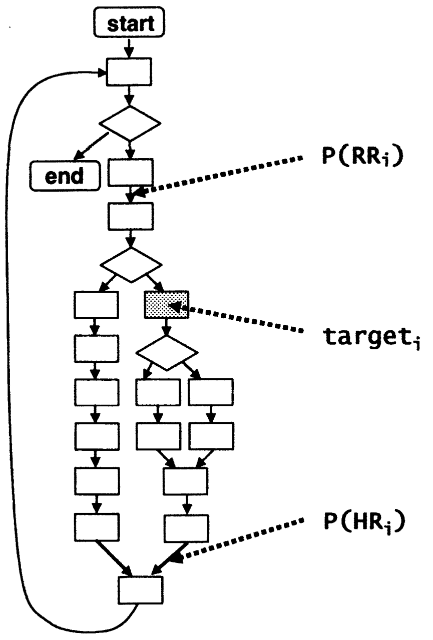


Fig. 3. Example of $target_i$, $P(RR_i)$, and $P(HR_i)$ that satisfy four conditions.

Then, a procedure for choosing $P(RR_i)$ and $P(HR_i)$ satisfying conditions 1 to 4 is presented.

(1) The set T_u of paths (routes without node duplication) from *start* to $target_i$ is determined.

(2) In the nodes that are common to all paths $t \in T_u$ determined in (1), the set N_u whose incoming and outgoing orders are both 1 is determined. It is assumed that $target_i \notin N_u$. If $N_u = \emptyset$, define $target_i$ anew and go back to (1).

(3) A node $n_u \in N_u$ is selected at random. The incoming or outgoing edge of n_u is defined as $P(RR_i)$. Similarly,

(4) The set T_l of paths (route without node duplication) from $target_i$ to *end* is determined.

(5) In the nodes contained in common to all paths $t \in T_l$ determined in (4), the set N_l of paths whose incoming and outgoing orders are both 1 is determined. It is assumed that $target_i \notin N_l$. If $N_l = \emptyset$, define $target_i$ anew, and go back to (1).

(6) A node $n_l \in N_l$ is selected at random. The incoming or outgoing edge of n is defined as $P(HR_i)$.

(Step 2) Determination of dummy instruction

An arbitrary instruction with the same instruction length as $target_i$ is selected and is defined as the dummy instruction $dummy_i$. An example is presented below in which the operation code composing $target_i$, or one of the operands, is modified for 1 byte, and is used as $dummy_i$. Consider the following $target_i$.

(hexadecimal machine language representation)	`03 5D F4`
(assembly language representation)	`addl -12(%ebp), %ebx`

By modifying operation code `03` to `33` in this $target_i$, the following $dummy_i$ is composed.

(hexadecimal machine language representation)	`33 5D F4`
(assembly language representation)	`xorl -12(%ebp), %ebx`

By modifying the operand F4 to `FA` in $target_i$, the following $dummy_i$ is produced:

(hexadecimal machine language representation)	`03 5D FA`
(assembly language representation)	`addl -6(%ebp), %ebx`

(Step 3) Generation of self-modification routine

The self-modification routines RR_i and HR_i are generated by the following procedure.

(1) Label L_i is inserted immediately before $target_i$.[*] Using label L_i, $target_i$ can be referred to indirectly.

(2) Using L_i, a series of instructions for the rewriting of $dummy_i$ to $target_i$ is constructed and is defined as RR_i.

(3) Using L_i, a series of instructions for the rewriting of $target_i$ to $dummy_i$ is constructed and is defined as HR_i.

An example is presented below. `addl -12(%ebp), %ebx` is defined as $target_i$, and `xorl -12(%ebp), %ebx` is defined as $dummy_i$. Label `L1` is inserted into $target_i$.

```
L1:addl -12(%ebp), %ebx
```

Next, RR_i is generated. RR_i has the function of modifying the first Byte `33` of the instruction at `L1` to `03`:

```
movb $0x03, L1
```

The effect of this small assembly routine composed of the above instruction is that the content of the address indicated by `L1` is to be overwritten by the immediate value `03` (hexadecimal). When RR_i is executed, $dummy_i$ is rewritten as $target_i$.

Similarly, HR_i is generated. HR_i has the function of modifying the first Byte `03` of the instruction at `L1` to `33`.

```
movb $0x33, L1
```

[*]A label is a name in assembly language which indicates the position of the instruction (memory address) in the program.

When HR_i is executed, $target_i$ is rewritten as $dummy_i$.

(Step 4) Write-in of dummy instruction and insertion of self-modification routine

The dummy instruction $dummy_i$ generated in Step 2 is written over $target_i$ determined in Step 1. By this process, the program before execution enters a state in which $target_i$ is camouflaged by $dummy_i$. Then the self-modification routines RR_i and HR_i which were generated in Step 3 are inserted into $P(RR_i)$ and $P(HR_i)$, respectively.

(Step 5) Complication of self-modification routine

The self-modification routine has the property that the address of $target_i$ in the program area is indicated by the label (immediate value address), and the content is rewritten. Consequently, there is a danger that an attacker may ascertain the position through the (static) analysis, and may identify the position of $target_i$. Consider, for example, the case in which the `movb` instruction in the program contains the immediate address indicating the program area as the second operand. Then, that `movb` instruction may be inferred to be a self-modification routine.

In order to make static analysis difficult, the self-modification routine is complicated. For example, the fact that there is no write-in into the program area may be disguised by operating on the label. Or, the identification of the self-modification routine by the static pattern is made more difficult by the use of conventional techniques such as obfuscation of machine language instructions [20] and mutation [11]. An example of the modification of `movb $0x03,L1` is as follows:

```
movl $L1 + 1250, %eax
subl $1250, %eax
movb $0x03,(%eax)
```

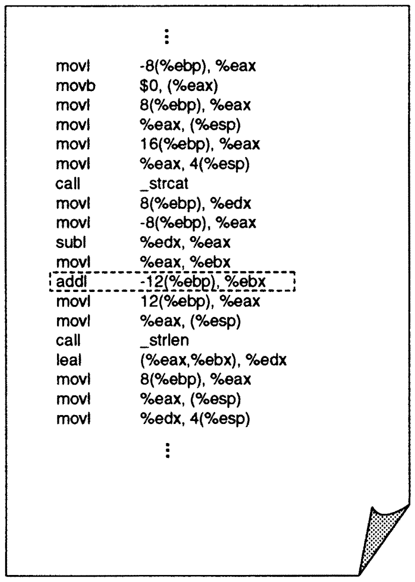
`L1` does not appear in the binary program obtained by assembling the above assembly routine (the value obtained by adding 1250 to `L1` appears). This makes it difficult to identify the address `L1` (the position of $target_i$) by static analysis. The address obtained by adding 1250 to `L1` does not necessarily indicate the program area. Furthermore, the second operand of the `movb` instruction is not the immediate address, but the address indicated by the register `%eax`. It is difficult to determine its value by static analysis. By combining the above processing with obfuscation and mutation, an attack by pattern matching and address analysis will be made more difficult.

(Step 6) Iteration of above steps

The processes from Step 1 to Step 5 are repeated. The number of camouflaged instructions is increased by each iteration. As will be discussed in Section 4, the increase in the number of camouflaged instructions and degradation of the execution efficiency are in a trade-off relation. It is thus desirable to specify the number of iterations with reference to the required degree of protection and the acceptable degradation of execution efficiency.

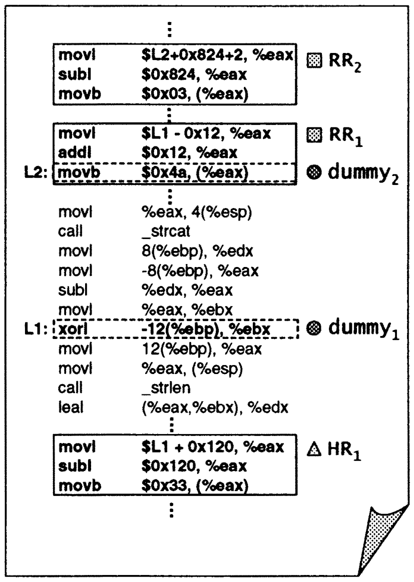
2.5. Construction example of camouflaged program *M*

Figure 4 shows an example of a camouflaged program. (a) is the original program and (b) is the camouflaged



```
            :
movl      -8(%ebp), %eax
movb      $0, (%eax)
movl      8(%ebp), %eax
movl      %eax, (%esp)
movl      16(%ebp), %eax
movl      %eax, 4(%esp)
call      _strcat
movl      8(%ebp), %edx
movl      -8(%ebp), %eax
subl      %edx, %eax
movl      %eax, %ebx
addl      -12(%ebp), %ebx
movl      12(%ebp), %eax
movl      %eax, (%esp)
call      _strlen
leal      (%eax,%ebx), %edx
movl      8(%ebp), %eax
movl      %eax, (%esp)
movl      %edx, 4(%esp)
            :
```

(a) Original Program



```
               :
    movl      $L2+0x824+2, %eax    ▨ RR2
    subl      $0x824, %eax
    movb      $0x03, (%eax)
               :
    movl      $L1 - 0x12, %eax     ▨ RR1
    addl      $0x12, %eax
L2: movb      $0x4a, (%eax)        ● dummy2
               :
    movl      %eax, 4(%esp)
    call      _strcat
    movl      8(%ebp), %edx
    movl      -8(%ebp), %eax
    subl      %edx, %eax
    movl      %eax, %ebx
L1: xorl      -12(%ebp), %ebx      ● dummy1
    movl      12(%ebp), %eax
    movl      %eax, (%esp)
    call      _strlen
    leal      (%eax,%ebx), %edx
               :
    movl      $L1 + 0x120, %eax    △ HR1
    subl      $0x120, %eax
    movb      $0x33, (%eax)
               :
```

(b) Camouflaged Program

Fig. 4. Example of a camouflaged program.

program. The procedure for deriving (b) from (a) is as follows.

In the first camouflaging process, the instruction [`add -12(%ebp), %ebx`] in the dotted frame in Fig. 4(a) is selected as *target*$_1$, and is overwritten by *dummy*$_1$ (`xorl -12(%ebp), %ebx`), as shown in Fig. 4(b). Then, the self-modification routines RR_1 and HR_1 for *target*$_1$ are generated and inserted. In the second camouflaging process, one of the instructions composing RR_i [`movb $0x03, (%eax)` at the end of the first camouflaging process] is selected as *target*$_2$ and is overwritten by *dummy*$_2$ [`movb $0x4a, (%eax)`] as shown in Fig. 4(b). Then, the self-modification routines RR_2 and HR_2 for *target*$_2$ are generated and inserted.

In this case, part of RR_1 is rewritten by *dummy*$_2$. Consequently, in order to ascertain the original instruction for *dummy*$_1$, not only RR_1 but also RR_2 must be found. In the Appendix a simple program containing a conditional branch is presented, together with a listing of the camouflaged program.

3. Discussion of Difficulty of Analysis

3.1. Assumed analysis procedure

Consider the case in which the attacker described in Section 2.1 attempts an analysis of the secret part $C(M)$ of M. The following attack procedure is assumed. The goal of the analysis is defined as understanding $C(M)$ correctly. In order to understand $C(M)$ correctly, the original instruction corresponding to each of the dummy instructions contained in $C(M)$ must be ascertained. For this purpose, the restoring routine for each dummy instruction contained in $C(M)$ must be found from the whole program.

There are two methods of analysis that the attacker can apply, static analysis and dynamic analysis. Static analysis is a method of analysis without running the program which is the object of analysis. A typical approach is to restrict the range of $C(M)$ by keyword retrieval, pattern matching, and other techniques, so as to understand $C(M)$. Since the analysis is concentrated on $C(M)$ without considering the whole of program M, the cost of analysis is generally lower than that of the dynamic analysis discussed later, and the method is widely applied. The first objective of the proposed method is to make static analysis difficult.

On the other hand, dynamic analysis is performed while running the program which is the object of analysis. The attacker runs M using tools such as a debugger, and tries to identify and understand $C(M)$ based on the output information from the tool. By dynamic analysis, the attacker can completely track the execution of M. However, since the analysis depends on the input and the whole program M must be run, the cost of analysis increases very rapidly as the scale of M is enlarged.

Furthermore, debugging information is generally deleted from commercial programs, or features such as the inhibition of unintentional execution are included. Consequently, it is not necessarily true that dynamic analysis can be applied to any program. It is possible at relatively low cost to preserve a snapshot at an arbitrary point of the executed program, that is, the content of the object program loaded into memory at any point during execution, in order to facilitate static analysis. For this purpose, there must be a mechanism to prevent the invalidation of protection even if several snapshots are acquired.

The next section discusses the security of M against each method of analysis.

3.2. Security against static analysis

In order to investigate the security of M against static analysis, the probability that the attacker can correctly understand the secret part $C(M)$ is formulated.

Consider the situation in which M contains only one dummy instruction *dummy*$_i$. In order for the attacker to correctly understand an arbitrary code block $D(M)$ of length m in M, the following event E_i must apply.

E_i: *dummy*$_i$ does not exist in $D(M)$, or *dummy*$_i$ exists in $D(M)$ and RR_i exists in $D(M)$.

When *dummy*$_i$ does not exist in $D(M)$ (i.e., there is no camouflage at all), the attacker can directly track $D(M)$ and can easily understand the original behavior of $D(M)$. When *dummy*$_i$ is present in camouflaged form in $D(M)$, but its restoring routine RR_i also exists in $D(M)$, *target*$_i$ can be identified by analysis of RR_i, and the original behavior of $D(M)$ can be discovered.

Let the number of instructions in M be L. When *dummy*$_i$ and RR_i are selected at random in M, the probability $P(E_i)$ that E_i is valid is expressed as follows:

$$P(E_i) = \frac{L-m}{L} + \frac{m}{L} \times \frac{m}{L} = \frac{(L-m)^2 + Lm}{L^2}$$

Then, consider the case in which n dummy instructions (*dummy*$_1$, . . . , *dummy*$_n$) are contained in M, and E_i must be valid for any $i (1 \le i \le n)$. The probability $P(Success, D)$ that the analysis of $D(M)$ succeeds is roughly expressed as follows:

$$P(Success, D) = \left(\frac{(L-m)^2 + Lm}{L^2}\right)^n$$

Figure 5 shows the curve representing the relation between $P(Success, D)$ and n. The horizontal axis is the number of camouflaged instructions n in M, and the vertical

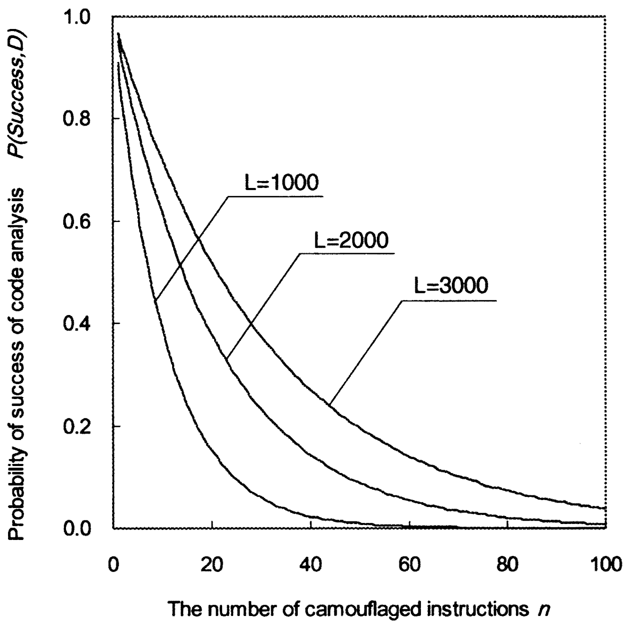


Fig. 5. Probability of success of code analysis ($m = 100$).

axis is the probability of successful analysis *P(Success, D)*. The number of instructions m in $D(M)$ is set as 100. The number of instructions in M is varied as 1000, 2000, and 3000. The result for each of these is shown. It is evident from Fig. 5 that as the number of dummy instructions n is increased (and thus the extent of camouflage is raised), the probability of successful analysis for $D(M)$ approaches 0.

When the secret part $C(M)$ agrees with (or is contained in) the code block $D(M)$ which is arbitrarily selected by the attacker, this implies that the static analysis of M is a success. Since the identification of $C(M)$ depends on the skill of the attacker, formulation by the theory of probability is difficult. Letting, as an assumption, the probability that $C(M)$ is contained in $D(M)$ be X, the probability of successful analysis *P(Success)* is expressed as

$$P(Success) = X \times P(Success, D) = \left(\frac{(L-m)^2 + Lm}{L^2}\right)^n X$$

Based on the above formulation, it is evident that in order to increase the probability of successful static analysis by the attacker, it is necessary for him either to increase X by skillfully locating $C(M)$, or to enlarge the size m of the analyzed part $D(M)$. On the other hand, the user of the proposed method can easily control *P(Success)* by increasing the number of camouflaged instructions n.

In the above discussion, *P(Success)* increases with the size of L. This is because $dummy_i$ is chosen at random in the formulation of the event E_i, which can prevent $dummy_i$ from being contained in $C(M)$. However, when the user knows the position of $C(M)$ beforehand, $P(E_i)$ can be decreased by inserting $dummy_i$ into $C(M)$, or by increasing the distance between RR_i and $dummy_i$ so that it is larger than the expected m. Thus, the probability of successful analysis can be decreased.

On the other hand, when the user does not know exactly the position of $C(M)$, or wants to decrease the probability X that the attacker can locate $C(M)$, it will be effective to divide M into L/m blocks and to insert a constant number of camouflaged instructions in each block. This measure makes the analysis difficult, no matter which block $D(M)$ the attacker subjects to analysis, since the camouflaged instructions are distributed uniformly.

3.3. Security against dynamic analysis

When M is stopped at a point during execution with a debugger, some of the dummy instructions in $C(M)$ may be in the state of being rewritten as the original instructions. If the attacker acquires a snapshot and observes the part corresponding to $C(M)$ in the program loaded in memory, some of the original instructions can be determined. This poses a danger that $C(M)$ may be correctly understood.

However, in this process it is difficult to know the original content of all dummy instructions present in $C(M)$. The reason is as follows. Since the restoring routine used to rewrite the dummy instruction in $C(M)$ is scattered over the whole program, various parts of the program must be executed in order to execute all of these routines. Unless the whole program is understood, this process has a high cost. Furthermore, when the hidden routine is executed, the instruction that was present in the original content is again overwritten by the dummy instruction. Consequently, even at the point immediately before the end of the program, the attacker cannot acquire a snapshot in which most of the instructions have been restored to the original instructions.

However, especially when fewer dummy instructions are present in $C(M)$, dynamic analysis can be an effective mode of attack. Consequently, it is desirable to use other techniques to make dynamic analysis difficult by preventing the operation of a debugger that uses interrupts and other instructions. This will improve the security of M against dynamic analysis.

4. Case Studies

4.1. Outline

This section describes the measurement process and the results for the following three items when the proposed method is applied to software.

(1) The distance between the target instruction and the restoring routine

(2) The change of the file size (size overhead)

(3) The change of the execution time (performance overhead)

We used the tool `ccrypt`, which encrypts and decrypts files, as the software with which to test the proposed method. This program is open-source software under the GPL license.*

The authors experimentally constructed a system in which the program was camouflaged by the proposed method [14]. Using that system, the proposed method was applied to the target program by the following procedure.

(1) The source file $s_1, s_2, \ldots, s_n$ in the C language was compiled and the original assembly file $a_1, a_2, \ldots, a_n$ was obtained.

(2) Each of $a_1, a_2, \ldots, a_n$ was camouflaged, and the camouflaged assembly file $a'_1, a'_2, \ldots, a'_n$ was obtained.

(3) $a'_1, a'_2, \ldots, a'_n$ were assembled and the execution modules $o_1, o_2, \ldots, o_n$ were obtained.

(4) $o_1, o_2, \ldots, o_n$ are linked and the executable file p is obtained.

In each trial, it is verified that the executable file p operates correctly.

In executable files running under Windows (such as the Microsoft Portable Executable format), enabling/disabling of writing to the code area is controlled by a flag in the section header in the file [16]. When the proposed method is applied, it is necessary to make the code area rewritable during execution by setting the flag beforehand.

The computer used in the experiment had Windows XP as the OS, a main memory size of 512 Mbyte, and a Pentium 4 CPU (clock frequency 1.5 GHz, primary trace cache 12kμOps, primary data cache of 8 kbyte, secondary cache 256 kbyte).

4.2. Distance between target instruction and restoring routine

Figure 6 shows the distribution of the dummy instruction and the restoring routine for a camouflaged assembly language file. The file has 1490 lines and 947 instructions before camouflaging. One hundred thirty instructions are camouflaged. The vertical axis of the figure is the line number, and the horizontal axis is the line number modulo 30. By adding the value on the horizontal axis to the value on the vertical axis, the line number containing the instruction or the restoring routine is obtained. It is evident from Fig. 6 that the target instructions and restoring routines are scattered over the whole program.

* http://quasar.mathstat.uottawa.ca/~selinger/

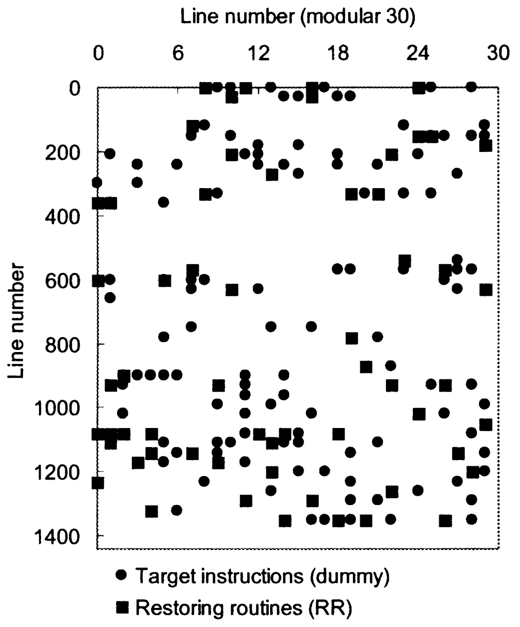


Fig. 6. Distribution of target instructions and restoring routines.

Table 1 shows the average, the maximum, the minimum, and the standard deviation of the distance between the target instruction and the restoring routine.

It can be seen from Table 1 that in order to ascertain whether an instruction in the program is a camouflaged instruction, the restoring routine, which is at a distance of 151 instructions away on average and 611 instructions away at the maximum, must be located. Since this program is camouflaged at a rate of 1 instruction in each 7 instructions, a large number of camouflaged instructions will be encountered in the search for the restoring routine. It can also happen, as was discussed in the example in Section 2.5, that instructions constituting the restoring routines are themselves camouflaged. Thus, it is likely that the cost of the analysis intended to find the restoring routine will be high.

Table 1. Distance between target instructions and restoring routines

Distance (instructions)	Average	Maximum	Minimum	
Standard deviation	151	611	1	192

Since the minimum distance is 1 instruction, it can be seen that there is a case in which the target instruction is adjacent to the restoring routine. Since the position of the target instruction and the insertion position of the restoring routine are selected at random from the candidates, such a case can occur. It is left as a future problem to improve the algorithm for choosing the insertion position, so that the target instruction and the restoring routine are more than a certain distance apart.

4.3. Size overhead

Examining the file size of the camouflaged program, it can be seen that the file size increases in proportion to the number of camouflaged instructions. On average, each time the number of camouflaged instructions is increased by 100, the file size is enlarged by approximately 2.4 kbyte. This increase in file size is due to the increase in the number of inserted self-modification routines as the number of camouflaged instructions is increased.

Noting that the capacity of secondary memory devices is currently increasing, the enlargement of file size will not be a serious problem. However, in an environment where the file size is severely limited it may happen that the increase of the file size must be minimized. It is possible to deal with such a situation by adjusting the number of camouflaged instructions so that the file size stays within the permissible range.

4.4. Performance overhead

The time required for the camouflaged `ccrypt` to encrypt a 100-kbyte text file was measured 10 times in each session while varying the number of camouflaged instructions. The execution time was measured as the difference in the elapsed time of the system clock from immediately before the start of the camouflaged program to immediately after the termination of the program. The elapsed time of the system clock was acquired by using the `clock` instruction in C.

Figure 7 shows a plot of the results of execution time measurement. The horizontal axis shows the number of camouflaged instructions and the vertical axis shows the average program execution time and the camouflage ratio (indicate by the bar). The camouflage ratio is the fraction of the program that is camouflaged.

It can be seen from Fig. 7 that the average execution time increases with the number of camouflaged instructions. When 500 instructions are camouflaged, the average execution time is approximately 2.9 seconds. This is approximately 48 times the execution time (approximately 0.06 second) when no instruction is camouflaged. There are three possible reasons for this increase in execution time.

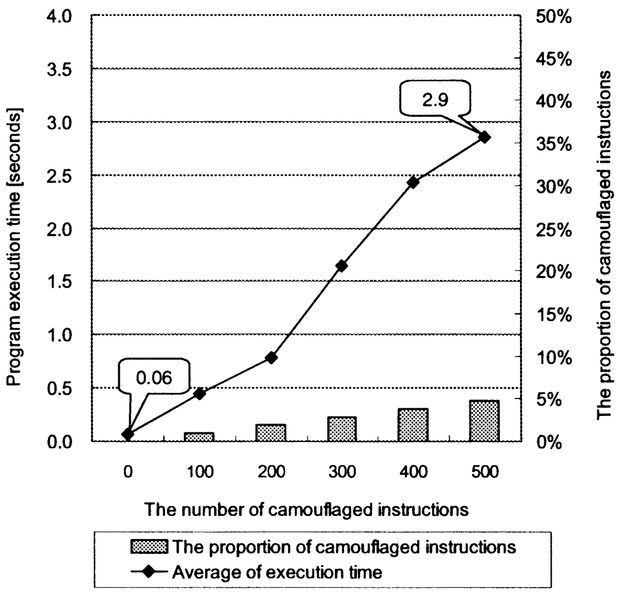


Fig. 7. Impact on program execution time.

(1) Inserting self-modification routines increases the number of instructions to be executed.

(2) Each time a self-modification routine writes code cached in the CPU, the corresponding cache line is invalidated [10].

(3) Self-modification mechanism increases the frequency of prediction failure of conditional branches in the CPU.

Increased execution time may or may not be a disadvantage. Excessive camouflaging is not recommended, for example, for an algorithm that considers the next move in a game such as shogi or chess, or for an algorithm which must operate in real time, such as a streaming playback routine for speech.

For a program in which the user is restricted by means of password authentication, on the other hand, the proposed method may be used in order to complicate the analysis of the password check routine. In such a case, if the method is applied only to the password checking part of the program, the original functioning is not degraded, except that a longer time is required for the password check. Thus, execution time overhead will not present a major problem. Location and extent of camouflage should be chosen according to the properties and purpose of the program or module to which the proposed method is applied.

5. Related Studies

The self-modification mechanism itself has long been known. One of its purposes is to reduce the program size and the required memory capacity in execution [8]. Another purpose is to protect programs, as in this study, in which a program is encrypted and decrypted by self-modification [4, 7, 12]. In the latter case, the specified range of the program is encrypted beforehand and is decrypted by self-modification during execution. The instructions are encrypted again if necessary. This approach is similar to the proposed method in the sense that the program is rewritten during execution, but differs in the following respects.

- It is not easy to identify the position of a camouflaged instruction by static analysis, since it cannot be distinguished from other instructions. On the other hand, the range of the part of the program which is encrypted may be easily identified, since it has features different from the other parts (such as the absence of instruction sequences and impossibility of disassembly).
- The restoring routine to resolve the camouflage is a very ordinary short routine in main memory, with a length of 1 to several bytes. In addition, since such routines are scattered through the program, it is not easy to identify their positions by static analysis. On the other hand, the restoring routine has a large size, and it is not easy to hide the identification cue. In particular, when the whole program is encrypted, the decryption starts immediately at the start of the program execution, which makes it easy to identify the restoring routine.

The approach in which an instruction which has been overwritten by a different content beforehand is replaced by the original instruction at the time of execution has been considered in the past for software protection. However, in the past approach, an operator with sufficient knowledge, technique, and assembler resources, had to handle the protection process manually. In contrast, this paper proposes and evaluates a systematic (formulated) method of protecting software by self-modification. An automatic protection mechanism is achieved by the proposed method. It is also made possible to flexibly adjust the trade-off (degree of camouflaging) between the degree of protection and the overhead.

Other methods of making program analysis difficult have included many methods of program obfuscation [5, 9, 19–21]. Obfuscation is a technique in which a given program is converted to a program which is more difficult to analyze (more complex) without modifying its specifications. The behavior of the obfuscated program can be understood correctly, even if partially, by spending a long time. On the other hand, in a program which is protected by the proposed method, it is difficult to determine, even partially, when an instruction with different content from the original instruction is present (unless the restoring routine has been identified), even if a long time is spent. This is the difference between obfuscation and the proposed method.

The proposed method is not a technique which is controversial with respect to encryption or obfuscation. Consequently, the analysis of the program is made still more difficult by combining the proposed method with these approaches. When the method is combined with obfuscation, for example, the result is a program which cannot be analyzed successfully unless the following two stages are accomplished.

(1) The uncamouflaged state is obtained.
(2) The obfuscated program is understood.

It should be noted, however, that the program size or the execution time may be further increased by the combined use of these approaches.

6. Conclusions

This paper has proposed a systematic method of making program analysis difficult by camouflaging instructions. When an attacker attempts a static analysis of a part of the program which contains camouflaged instructions, he cannot understand the original behavior of that part correctly unless he locates the restoring routine.

To investigate the difficulty of analysis of the camouflaged program, the probability that the attacker can correctly understand the secret part of the program was analyzed. Based on the resulting equation, it is concluded that the range of analysis must be enlarged in order to correctly understand the camouflaged program.

As a case study, a program (`ccrypt`) was camouflaged. The distance between the target instructions and the restoring routine, the file size, and the execution time overhead were measured. When 130 instructions in 947 were camouflaged, the average distance between the target instruction and the restoring routine was 151 instructions. Since many camouflaged instructions may occur between the target instruction and the restoring routine, it is likely that a costly analysis will be required in order to find the restoring routine. As regards overhead, it was found that the file size and the overhead of execution time are increased with an increasing number of camouflaged instructions. It is desirable to choose the position and extent of camouflage according to the properties and purposes of the program or module to which the method is applied.

Problems left for the future are as follows. It will be useful to improve the algorithm for choosing the insertion position of the self-modification routine so that the target instruction and the restoring routine are more than a certain distance apart. We also plan to improve the system in order to reduce the execution time overhead, so self-modification is handled from the viewpoint of the pipeline function and the branch prediction function of the CPU.

REFERENCES

1. Cerven P. Crackproof your software. No Starch Press; 2002.
2. Chang H, Atallah M. Protecting software codes by guards. Proc Workshop on Security and Privacy in Digital Rights Management 2001, LNCS Vol 2320, p 160–175, Springer-Verlag, 2001.
3. Chow S, Eisen P, Johnson H, van Oorschot PC. A white-box DES implementation for DRM applications. Proc 2nd ACM Workshop on Digital Rights Management, p 1–15, 2002.
4. Cohen FB. Operating system protection through program evolution. Comput Secur 1993;12:565–584.
5. Collberg C, Thomborson C. Watermarking, tamper-proofing, and obfuscation—tools for software protection. IEEE Trans Softw Eng 2002;28:735–746.
6. Funamoto S. Anatomy of protection technology. Subarusha; 2002.
7. Grover D (editor). The protection of computer software: Its technology and applications. Cambridge University Press; 1989.
8. Hidaka T. Mysteries of the Z80 machine. Keigaku Shuppan; 1989.
9. Hohl F. Time limited blackbox security: Protecting mobile agents from malicious hosts. In: Vigna G (editor). Mobile agents security. LNCS Vol. 1419, p 92–113, Springer-Verlag, 1998.
10. Intel Co. IA-32 Intel architecture software developer's manual vol. 3. System programming guide. Chapter 9, p 18, http://www.intel.co.jp/
11. Irwin J, Page D, Smart NP. Instruction stream mutation for non-deterministic processors. Proc ASAP2002, p 286–295.
12. Ishima H, Saito K, Kamei M, Shin Y. Tamper resistant technology for software. Fuji Xerox Tech Rep, No. 13, p 20–28, 2000.
13. Kanzaki Y, Monden A, Nakamura M, Matsumoto K. Prevention of program analysis by replacement of instruction code in execution. Tech Rep IEICE ISEC2002-98, Dec. 2002.
14. Kanzaki Y. Tool for program camouflage. http://se.aist-nara.ac.jp/rinrun/
15. Kanzaki Y, Monden A, Nakamura M, Matsumoto K. Exploiting self-modification mechanism for program protection. Proc 27th IEEE Computer Software and Applications Conference, p 170–179, Dallas, 2003.
16. Levine JR. Linkers and loaders. Morgan Kaufmann; 2000. p 75–83.
17. Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly. Proc 10th ACM Conference on Computer and Communications Security, p 290–299, 2003.
18. Monden A, Monsifrot A, Thomborson C. Obfuscated instructions for software protection. Information Science Technical Report, NAIST-IS-TR2003013, Graduate School of Information Science, Nara Institute of Science and Technology, 2003.
19. Monden A, Takada Y, Torii K. Methods for scrambling programs containing loops. Trans IEICE 1997;J80-D-I:644–652.
20. Murayama T, Mambo M, Okamoto H, Uematsu T. Obfuscation of software. Tech Rep IEICE 1995;ISEC-95-25.
21. Ogiso T, Sakabe Y, Soshi M, Miyaji A. Software obfuscation on a theoretical basis and its implementation. IEICE Trans Fundam 2003;E86-A:176–186.
22. Okamura H. The latest cases on the cyber law. Softbank Publ. Co.; 2000.
23. The United Kingdom Parliament. The mobile telephones (re-programming) bill. House of Commons Library Research Paper, No. 02-47, 2002.
24. Yamada H, Kawahara J. Current status of digital content protection and related issues. Toshiba Review 2003;58:2–7.

APPENDIX

A simple program containing a conditional branch and the camouflaged program are presented below

1. Original program (C language)

```
#include <stdio.h>
#define PASSNUM 13

int main() {
    int n;
    scanf("%d," &n);
    if(n!=PASSNUM) {
  printf("INVALID\n");
  return -1;
    }
    printf("OK\n");
    return 0;
}
```

2. Original program (assembly)

```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
LC2:
  .ascii "OK\12\0"
  .align 2
```

```
.globl _main
_main:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
  movl  %eax, 4(%esp)
  call  _scanf
  cmpl  $13, -4(%ebp)
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
  jmp  L9
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
L9:
  movl  -8(%ebp), %eax
  leave
  ret
```

3. Camouflaged program

```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
LC2:
  .ascii "OK\12\0"
  .align 2
.globl _main
_main:
  movl  $T2 + 0x824, %eax       # RR2
  subl  $0x824, %eax            # RR2
  movb  $0xeb, (%eax)           # RR2
  pushl  %ebp
  subb  $0x3d, T3 + 2           # RR3
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $T1 - 20 + 3, %eax      # RR1
  addl  $20, %eax               # RR1
T3:
  movb  $0x4a, (%eax)           # RR1 target3
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
  movl  %eax, 4(%esp)
  movl  $T3 - 0x08 + 2, %eax    # HR3
  addl  $0x08, %eax             # HR3
  movb  $0x4a, (%eax)           # HR3
  call  _scanf
T1:
  cmpl  $7, -4(%ebp)            # target1
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
T2:
  je  L9                        # target2
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
  movb  $0x74, T2               # HR2
L9:
  movl  -8(%ebp), %eax
  movl  $T1 + 0x120 + 3, %eax   # HR1
  subl  $0x120, %eax            # HR1
  movb  $0x07, (%eax)           # HR1
  leave
  ret
```

AUTHORS (from left to right)

Yuichiro Kanzaki (student member) received his B.S. degree in 2001 from the Department of Computer and Systems, Kobe University, completed the first half of his doctoral program in 2003, and is now in the second half at Nara Institute of Science and Technology. He is engaged in research on software protection. He is a student member of IEEE.

Akito Monden (member) received his B.S. degree in 1994 from the Department of Electrical Engineering, Nagoya University, completed the second half of his doctoral program in 1998 at Nara Institute of Science and Technology, and became a research associate in the Department of Information Science. He was a visiting researcher at the University of Auckland, New Zealand, in 2003–2004. His research interests are software protection, software metrics, and human interface. He holds a D.Eng. degree, and is a member of the Information Processing Society, Japan Software Science Society, IEEE, and ACM.

Masahide Nakamura (member) received his B.S. degree in 1994 from the Department of Information, Osaka University, and completed the second half of his doctoral program in 1999. He became a research associate at the Cybermedia Center in 2000. He has been a research associate in the Graduate School of Information Science, Nara Institute of Science and Technology, since 2002. His research interests are communications software, service competition, and software protection. He holds a D.Eng. degree, and is a member of IEEE.

Ken'ichi Matsumoto (member) received his B.S. degree in 1985 from the Department of Information, Osaka University. Before completion of his doctoral program, he became a research associate in the Department of Information in 1989. He was appointed an associate professor in the Graduate School of Information Science, Nara Institute of Science and Technology, in 1993, and has been a professor there since 2001. His research interests are software quality assurance, user interface, and software protection. He holds a D.Eng. degree, and is a member of the Information Processing Society, IEEE, and ACM.