Exploiting Self-Modification Mechanism for Program Protection

Yuichiro Kanzaki Akito Monden Masahide Nakamura Ken-ichi Matsumoto
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, 630-0192, JAPAN
{yuichi-k, akito-m, masa-n, matumoto}@is.aist-nara.ac.jp

Abstract

In this paper, we present a new method to protect software against illegal acts of hacking. The key idea is to add a mechanism of self-modifying codes to the original program, so that the original program becomes hard to be analyzed. In the binary program obtained by the proposed method, the original code fragments we want to protect are camouflaged by dummy instructions. Then, the binary program autonomously restores the original code fragments within a certain period of execution, by replacing the dummy instructions with the original ones. Since the dummy instructions are completely different from the original ones, code hacking fails if the dummy instructions are read as they are. Moreover, the dummy instructions are scattered over the program, therefore, they are hard to be identified. As a result, the proposed method helps to construct highly invulnerable software without special hardware.

1 Introduction

Software cracking has posed a serious problem for copyright protection of the software. A typical scenario is that a cracker analyzes a checking routine of copy protection, and then modifies the program so that the routine is nullified. Another scenario would be that, by analyzing a program of a business rival, one steals ideas and methods in the program. Crackers, who perform such illegitimate acts, have been creating a serious threat to the software industry so far.

In addition to the conventional stand-alone software, the cracking can cause a significant loss to the latest networked software technology. For example, [2] reports a risk of attack from crackers on the latest digital contents distribution system. Figure 1 shows an example of the system. The server sends compressed and encrypted digital contents to the client through the Internet. The client program that runs on a user's computer receives and saves encrypted data. The data is decrypted and decompressed when the user plays a

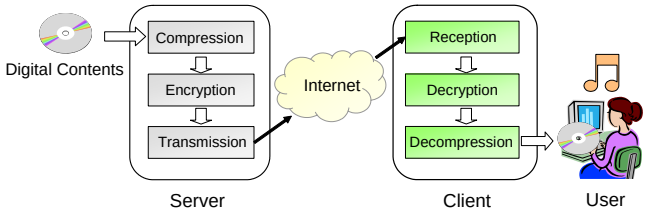

Figure 1. An example of digital contents distribution system

content. If a cracker succeeds in obtaining secret information(such as secret keys, special algorithm of decryption) by analyzing the decryption program, contents can be used illegally without authentication. Thus, the necessity of preventing acts of cracking is increasing, and the technology of software protection is a pressing issue to many software developers.

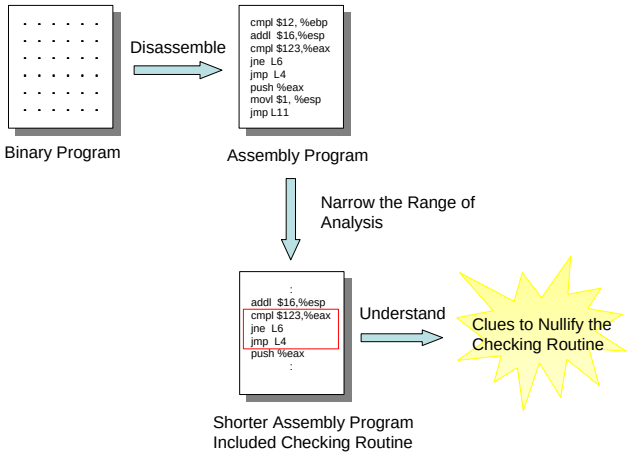

Figure 2. A scenario of obtaining clues to nullify the password checking routine

It is difficult to clearly answer how software is cracked,

since there are many ways to crack software programs. However, it is sure that a cracker has to *understand* a program in order to obtain clues to crack programs successfully.

Figure 2 shows a scenario, where a cracker obtains clues to nullify the password checking routine. First, the cracker disassembles the binary program into an assembly program to make it easy to understand. Then, the cracker often narrows the range of analysis to reduce costs for understanding the program, referring to resources of the program or something. Then, a program fragment containing the checking routine is identified and understood by the cracker. As a result, the checking routine may be canceled or cropped, then the program may be used illegally. As seen in this example, whenever a program is cracked, there must be a process where a cracker reads the program and tries to understand the program for obtaining clues to crack.

An effective solution to protect software against illegal acts of hacking is to increase costs for understanding the program. In this paper, we present a new method to increase the cost of understanding programs for protecting software. The key idea is to add a self-modification mechanism to the original program. By using the self-modification mechanism, we can camouflage the original instruction with a dummy instruction. If a cracker reads the camouflaged part as it is, he fails to understand the original instruction. We believe that the program protected by our method is quite hard to be understood, and that it is difficult for crackers to cancel the protection, since the dummy instructions are scattered over the program.

The rest of this paper is organized as follows: In Section 2, we review the related work. Section 3 describes the proposed method with an example. In Section 4, we examine how much overhead on the program size and the execution time is imposed by the proposed method. Finally, Section 5 concludes the paper with discussion and future work.

2 Related Work

Many methods for protecting software have been proposed so far. We categorize them into three types: *program obfuscation*, *program encryption*, and *program fragmentation*. All methods aim to effectively increase cost for understanding the program.

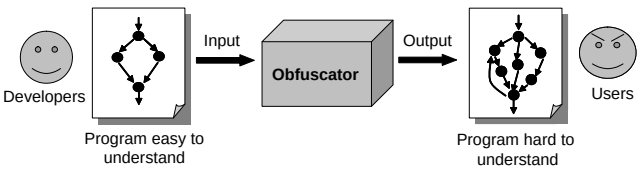

Figure 3. Program Obfuscation

Program obfuscation is to transform of a program into another program that is harder to understand. The expressions and procedures in the obfuscated program are much more complex than original one.

Figure 3 shows a basic concept of program obfuscation. When a developer inputs a program to an obfuscator, it transforms the program into one that is functionally identical to the original one. But, the resulting program is much more difficult for users to understand. A developer can reduce a risk of attack to his program by using an obfuscator. However, it is not easy to implement an obfuscator that universally works well for any kind of attacks. Many different approaches have been proposed. These include the use of complicating control structures [13] [18], replacing a high-level instruction with the combination of low-level instructions [17] , inserting dummy codes which does not affect the result [6] [7], transforming the data structures [8], changing method names in a program [23], changing reference or substitution of arrays and pointers [21] [24], etc.

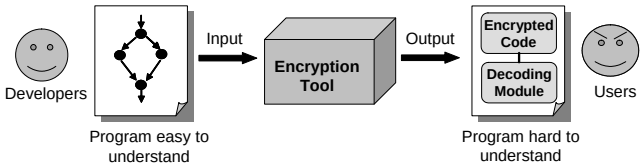

Figure 4. Program Encryption

Program encryption is a technique, which makes program code harder to understand with encryption. Figure 4 shows a basic concept of program encryption. When a developer inputs a program to an encryption tool, he can obtain an encrypted program. The encrypted program consists of two parts: the encrypted code and (non-encrypted) decoding modules which decode the program at run-time. A cracker is almost unable to understand the encrypted code. However, there is a risk that protection can be canceled easily by analyzing and modifying the decoding modules. The concrete methods for program encryption have been described in [1] [5] [10] [12] [14] [22], etc.

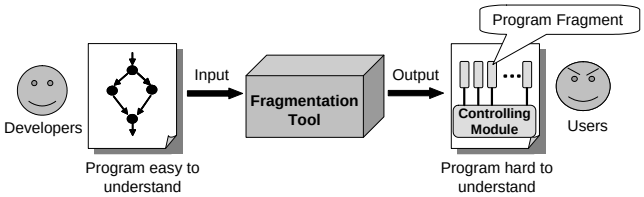

Figure 5. Program Fragmentation

Program fragmentation is a technique, which divides the program into many fragments and controls the execution sequence of them. Figure 5 shows a basic concept of program

fragmentation. When a developer inputs a program to an fragmentation tool, he can obtain a fragmented program. The fragmented program consists of two parts: fragments of the the original program and modules which control the execution sequence of them at run-time. It is difficult to understand the program for a cracker by reading the fragments. However, there is a risk that protection can be nullified by cracking the module. The concrete methods for program fragmentation have been described in [3] [4] [15] [20] , etc.

These previous work are promising under some attack models. However, we can say there is no conclusive method , and problems about software cracking are increasing [9][16][19]. Therefore, we propose a method based on a new approach to improve the present circumstances.

3 Protecting software by replacing instructions at run-time

3.1 Key idea

The key idea of the proposed method is to add *a self-modification mechanism* to the original program, to increase the cost of understanding the original program. In the self-modification mechanisms, an instruction p in the program replaces another instruction q in the same program with a different instruction r at run-time.

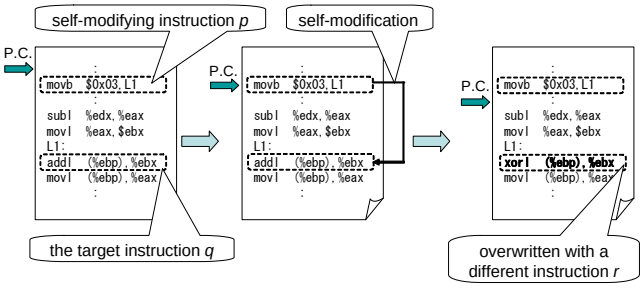

Figure 6. Self-modification mechanism

Figure 6 shows assembly codes to demonstrate self-modification [1] . P.C. represents a program counter. In this example, instructions "`movb $0x03,L1`", "`addl (%ebp),%ebx`" and "`xorl (%ebp),%ebx`" correspond to the above p, q and r, respectively. "`addl (%ebp),%ebx`" is a target instruction to be modified. This is overwritten with "`xorl (%ebp),%ebx`" at run-time by executing "`movb $0x03,L1`". Since the dummy instructions are completely different from the original ones, code hacking fails if the dummy instructions are read as they are.

[1] All assembly codes appeared in this paper are written in AT&T syntax.

Our approach primarily consists of two parts: Firstly, we camouflage many of the original instructions by dummy instructions. Secondly, to assure the correctness of the original program, we add self-modifying instructions that replace the dummy instructions with the original ones within a certain period of execution.

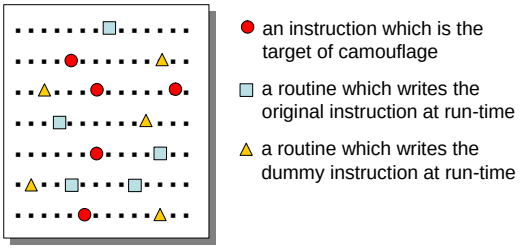

Figure 7. Concept of a program obtained by the proposed method

Figure 7 conceptually describes a program obtained by the proposed method. Multiple instructions in the program are selected as targets of camouflage(depicted by ●). They are overwritten with dummy instructions before execution. In the program, there are routines which write the original instruction at run-time(depicted by ■) and there are also routines that write back the dummy instructions(depicted by ▲). These pairs of routines are prepared exactly as many as the targets of camouflage. The original instruction in each target of camouflage is restored as the original instruction during an interval from a routine ■ to a routine ▲.

Thus, by using the self-modification mechanism, we can *camouflage* the original instruction with a dummy instruction. If a cracker reads the camouflaged part as it is, he fails to understand the original instruction. We believe that the program protected by our method is quite hard to be understood, and that it is difficult for crackers to cancel the protection, since the dummy instructions are scattered over the program. The protection will be much more invulnerable by using the proposed method together with other methods, such as program obfuscation, program encryption, program fragmentation.

3.2 Preliminary

Before explaining the proposed method, we give some definitions related to the self-modifying program. The *original code* O is an assembly program to be protected, which is given by the user.

A *target instruction* is an (original) instruction in O that the user wants to hide using the self-modification mechanism. Since the user can specify multiple target instructions to be camouflaged, let $target_i$ be the i-th target instruction.

For $target_i$, $dummy_i$ denotes a *dummy instruction* which overwrites $target_i$ as its camouflage.

A *restoring routine* is a set of instructions which uncamouflage an original target instruction hidden by a dummy instruction. On the contrary, a *hiding routine* is a set of instructions which camouflage a target instruction with a dummy instruction. Specifically, a restoring routine RR_i is a set of instructions that replace $dummy_i$ with $target_i$ at run-time. On the other hand, a hiding routine HR_i is a set of instructions that replace $target_i$ with $dummy_i$ at run-time.

Our key idea can be explained as follows. First, the user select a set of target instructions $target_i$'s from O. For each $target_i$, decide a certain $dummy_i$, and overwrite $target_i$ with $dummy_i$. Then, for $dummy_i$, make RR_i and insert it at some position in O, so that RR_i is executed before $dummy_i$. Similarly, make HR_i and insert it at some position in O, so that HR_i is executed after $dummy_i$. By doing this, $target_i$ appears instantly within a period between RR_i and HR_i. In any other periods, $target_i$ is hidden by $dummy_i$.

A *self-modifying code* M is the resulting program obtained by applying the above procedure for all $target_i$'s in O. A *camouflaged instruction* is an instruction in M which is modified by $target_i$ and $dummy_i$.

For $target_i$, there are basically several ways to determine $dummy_i$, RR_i and HR_i. The decision can be left to the users, or can be implemented by a random selection from appropriate candidates.

In the following subsections, we describe a systematic method to construct the self-modifying code M from a given original code O.

3.3 Outline of the proposed method

Figure 8 shows an outline of the proposed method. First, a user (e.g. a program developer) who uses the proposed system prepares an assembly program (original code) O to be protected. This is normally obtained by compiling a source program or by disassembling a binary program. Then, the proposed system adds the self-modification mechanism to the assembly program, so that the original program becomes hard to be analyzed. Finally, assembling an assembly program M, which is the output of the system, the user can obtain a self-modifying program in binary that is functionally equivalent to the original one, but which is much more complex for crackers to analyze.

The proposed system constructs a self-modifying program by following six steps:

(Step 1) Determining the positions of adding routines and an instruction which is the target of camouflage

(Step 2) Generating a dummy instruction to camouflage

(Step 3) Generating routines

(Step 4) Writing the dummy instruction and inserting the routines

(Step 5) Complicating the inserted routines

(Step 6) Repeating previous steps and constructing the self-modifying program

3.4 Procedure of constructing a self-modifying program

(Step 1) Determining the positions of adding routines and an instruction which is the target of camouflage

First, we determine the positions of $target_i$, RR_i and HR_i. Now we define that $P(target_i)$, $P(RR_i)$ and $P(HR_i)$ correspond to the position of instruction $target_i$, the position of inserting RR_i and the position of inserting HR_i, respectively.

First, $P(target_i)$ is randomly selected by the system. Then, $P(RR_i)$ and $P(HR_i)$ are determined that they will satisfy the following three conditions, which are necessary for a program not to cause malfunction by adding a self-modifying function.

1. $P(RR_i)$ must exist on every control flow path from the program entry to the $P(target_i)$.
2. $P(HR_i)$ must not exist on every control flow path from $P(RR_i)$ to $P(target_i)$.
3. $P(RR_i)$ must exist on every control flow path from $P(HR_i)$ to $P(target_i)$.

Figure 9 illustrates an example of determining them with a control flow graph of a program. There are basically several candidates for the positions of the routines. The decision is normally by a random selection.

(Step 2) Generating a dummy instruction to camouflage

We generate a dummy instruction $dummy_i$. Let us consider the following construction as $target_i$.

(Hex Representation)	`03 5D F4`
(Assembly Representation)	`addl -12(%ebp),%ebx`

Now, we want to decide a dummy instruction for this. A dummy instruction is obtained by changing the content of $target_i$, so that the operation code or the operand will be different. As for a possible candidate, we choose `xorl`, which is changed first byte from "`03`" to "`33`" , as shown below:

(Hex Representation)	`33 5D F4`
(Assembly Representation)	`xorl -12(%ebp),%ebx`

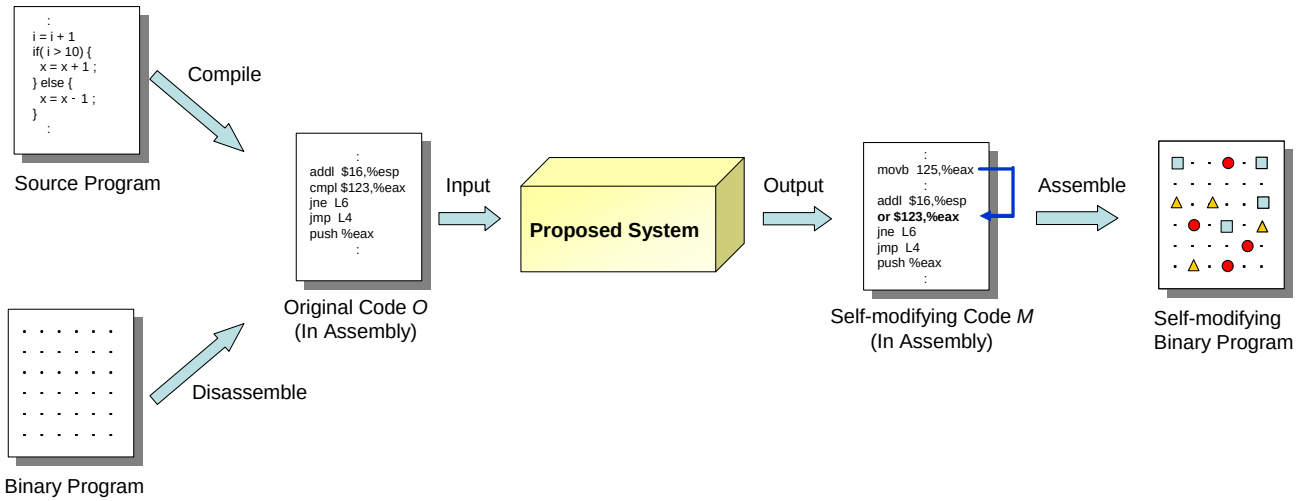

Figure 8. An outline of the proposed method

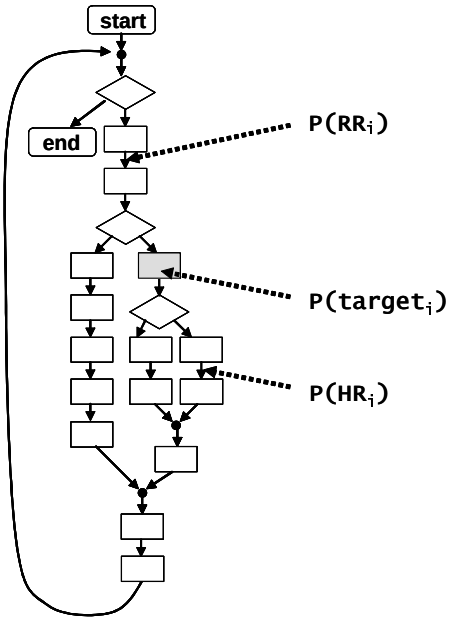

Figure 9. Determining the positions of an instruction to be camouflaged and adding routines

(Step 3) Generating routines

We generate routines, RR_i and HR_i. Both of them are described in assembly to be a part of the self-modifying code M. The role of RR_i is to transform $dummy_i$ into $target_i$ in $P(target_i)$ at run-time. In contrast, HR_i replaces $target_i$ with $dummy_i$ in $P(target_i)$ at run-time. We define routines which perform the above as the follows:

1. Insert a label L_i just before the $P(target_i)$.
2. Create instruction(s) to write some bytes for turning $dummy_i$into $target_i$ with L_i.
3. Create instruction(s) to write some bytes for turning $target_i$into $dummy_i$ with L_i.

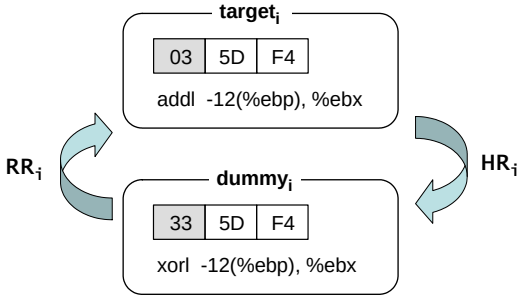

Figure 10. Restoring and hiding routines

With the example in (Step 2), the roles of RR_i and HR_i are shown in Figure 10. Specifically, RR_i is to change the first byte of the instruction in $P(target_i)$ from "`33`" to "`03`", and HR_i is to change the first byte of the instruction in $P(target_i)$ from "`03`" to "`33`".

When a label `L1` is inserted just before the $P(target_i)$, RR_i can be generated as below:

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

This small assembler routine means that the first byte of code where the label `L1` is pointing is overwritten with the immediate value "`03`" in hex. When this routine runs, The instruction in $P(target_i)$ is set to $target_i$ from $dummy_i$.

In the same way as RR_i, HR_i can be generated as below:

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

When this routine runs, the instruction in $P(target_i)$ is set to $dummy_i$ from $target_i$.

(Step 4) Writing the dummy instruction and inserting the routines

The dummy instruction $dummy_i$, which was generated in (Step 2), is overwritten to $P(target_i)$ determined in (Step 1). Also, RR_i and HR_i, which were generated in (Step 3), are respectively inserted into positions $P(RR_i)$ and $P(HR_i)$, determined in (Step 1).

(Step 5) Complicating the inserted routines

Crackers may identify the inserted routines easily since both of the routines are described so simply. Thus, it is desirable that the inserted routines are transformed so that they become more complicated. The example of complicating routines is described in subsection 3.5. Due to page limitation, we omit the description of methods for complicating routines in this paper.

(Step 6) Repeating previous steps and constructing the self-modifying program

Repeat from (Step 1) to (Step 5). A user can decide the number of repetition, according to the required protection level. It is recommended that the number is not too small, to achieve a certain level of the protection.

3.5 Example

In this subsection, we provide an example of constructing a self-modifying program according to the procedure described above. Here, we show from (Step 1) to (Step 5), which is one cycle of construction. Now given that original program O which is inputted to the system is such as shown in Figure 11.

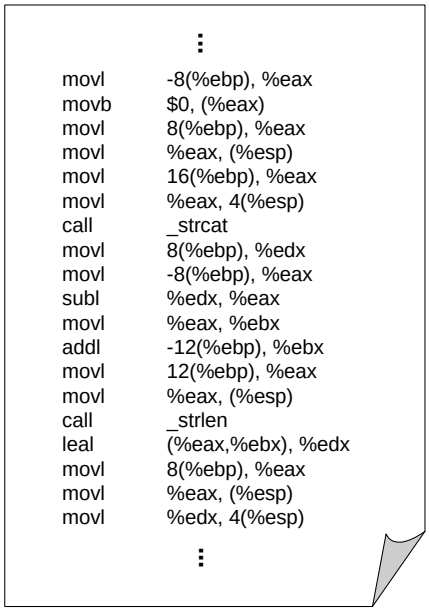

Figure 11. Original Assembly Program

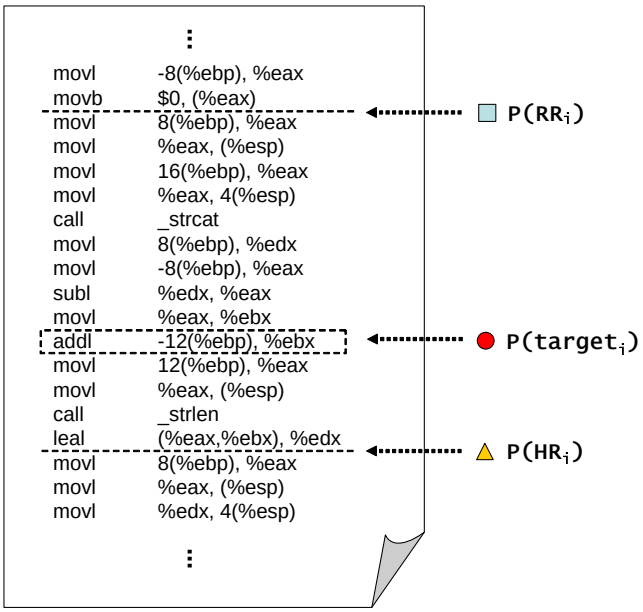

Figure 12. Determining the positions

(Step 1)

First, $P(target_i)$ is randomly selected by the system. Now given that "`addl -12(%ebp),%eax`", the instruction on the ninth line from the bottom, is selected as $P(target_i)$ (See Figure 12). Next, $P(RR_i)$ and $P(HR_i)$ are determined. In this example, they were determined such as shown in Figure 12. $P(RR_i)$ and $P(HR_i)$ satisfy the conditions described in subsection 3.4.

(Step 2)

We generate a dummy instruction $dummy_i$ by changing some bytes of $target_i$, "`addl -12(%ebp),%eax`". In this example, we generate "`xorl -12(%ebp),%eax`" as the dummy instruction $dummy_i$, by the same way described in subsection 3.4.

(Step 3)

We generate routines RR_i and HR_i. First, a label "`L1`" is inserted just before the $P(target_i)$. Then, RR_i is generated such as "`movb $0x03,L1`", and HR_i is generated such as "`movb $0x33,L1`", by the same way described in subsection 3.4.

(Step 4)

The $dummy_i$ ("`xorl -12(%ebp),%eax`") is written to $P(target_i)$, where "`addl -12(%ebp),%eax`" is. And the routines which were generated in (Step 3) are inserted into the each positions. Figure 13 shows the program after this step.

(Step 5)

We complicate the inserted routines, RR_i and HR_i. In this example, we note that the inserted routines are easy to identify since both of the routines describe the position of the target instruction in the same way. Thus, the routines are changed so that RR_i and HR_i use different labels("`A1`" and "`A2`") to point at the $P(target_i)$. Figure 14 shows the routines after this step. By this step, inserted routines and the target instruction become hardly to identify.

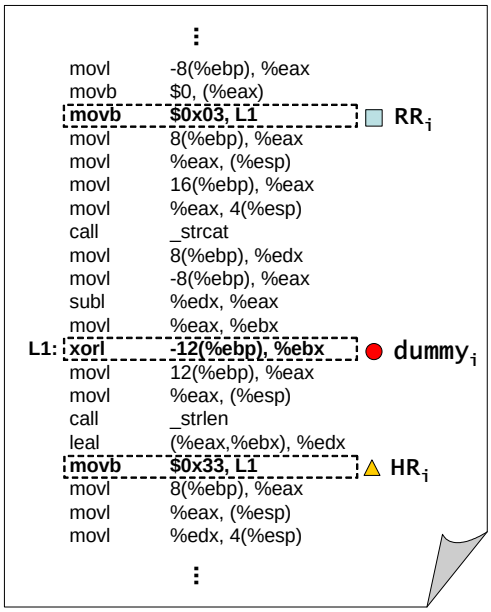

Figure 13. Writing the dummy instruction and inserting routines

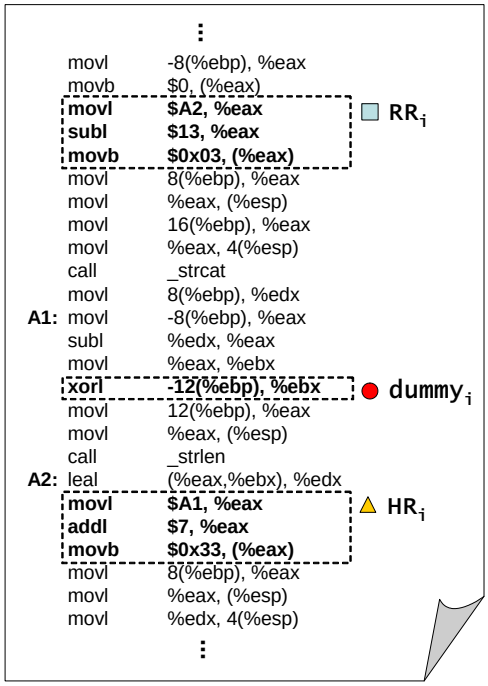

Figure 14. Complicated routines

4 Experiment

4.1 Overview of the experiment

In this section, we examine how much overhead on the program size and the execution time is imposed by the proposed method. We applied the proposed method to `gzip`, a well-known GNU utility for compressing and decompressing files [11].

We implemented a system based on the proposed method. By using the system, we camouflaged instructions in the program, and inserted restoring routines and hiding routines to `gzip`. Then, we measured the program size and execution time with different number of camouflaged instructions. The number of camouflaged instructions was varied from 200 to 1000 with an interval of 200.

4.2 Size overhead

The graph in Figure 15 illustrates the impact of the self-modification mechanism on program size. The horizontal axis represents the number of camouflaged instructions, while the vertical axes plot program size(depicted in a line) and the proportion of the camouflaged instructions to the total instructions in the original program. By the proportion of camouflaged instructions, we characterize the *degree* of the camouflage in the program.

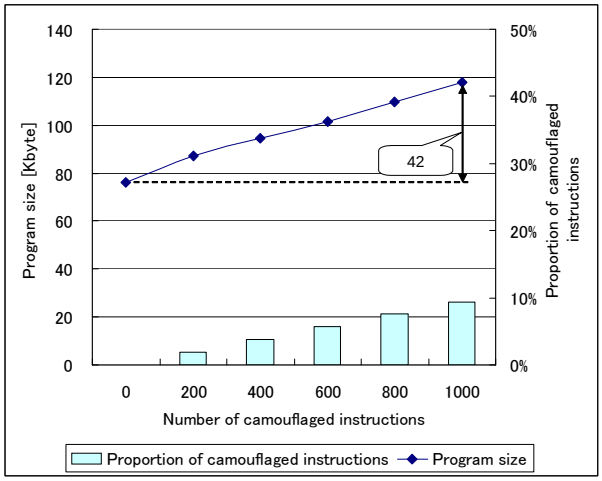

Figure 15. Impacts on program size

We can see, in this graph, that the program size increases in proportion of the number of camouflaged instructions. This is because new instructions for hiding and restoring routines are added when we camouflaged each instruction. Specifically, the program size increases about 4.2Kbyte as every 100 camouflaged instructions are embedded. Even when the number of camouflaged instructions is 1000, where about 9% of the total instructions are camouflaged, the overhead in program size is as small as 42Kbyte. Note that the overhead in program size imposed is *independent* of the size of the original program, since the number of instructions involved in the hiding/restoring routines do not depend on the original program.

4.3 Performance overhead

Next, we measure the performance of the modified programs. In the experiment, we used a 1 Mbyte text file, and measured the time taken for each (modified) `gzip` to compress the text file. Since the proposed system selects the position of the hiding/restoring routines at random(see Section 3), the execution time varies for every measurement. Therefore, we measured the execution time ten times, and calculated the average, minimum and maximum values.

Figure 16 illustrates the result on execution time. The horizontal axis represents the number of camouflaged instructions, while the vertical axes show the execution time (average drawn by a solid line, minimum and maximum depicted by dotted lines) and the proportion of the camouflaged instructions to the total instructions in the original program.

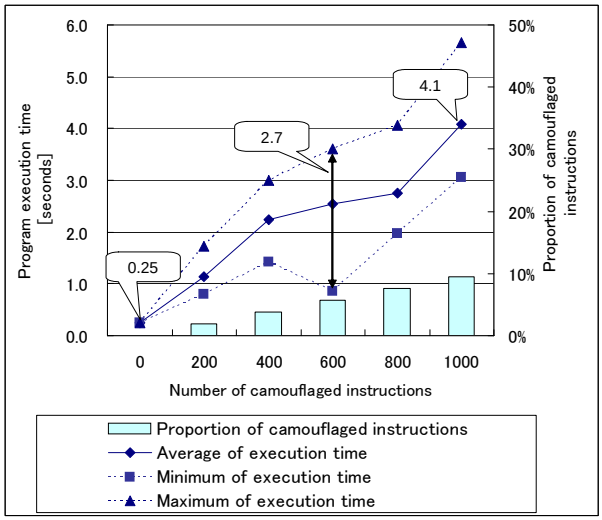

Figure 16. Impacts on program execution time

The execution time increases in the number of camouflaged instructions. When 1000 instructions are camouflaged, the average execution time is about 4.1 seconds, which is about 16 times as long as the original (0.25 seconds). We consider that the overhead is caused by the inserted hiding/restoring routines. Additionally, although not investigated in much detail, we guess that the self-modification mechanism imposes an extra overhead to CPU, due to architectural issues such as incoherence of cache memory, or failure of instruction pre-fetch.

From the lines of the minimum and the maximum, it can be seen the execution time varies much. For example, with 600 instructions camouflaged, the difference is 2.7 seconds. This fact implies that there could be a way to improve

the program performance. For instance, if we carefully insert the routines into positions that are not frequently executed(e.g. outside loops), then the overhead caused by the routines might be significantly reduced.

5 Discussion and concluding remarks

In this paper, we have presented a new method to protect software against illegal acts of hacking. The key idea is to add a self-modification mechanism to the original program, to increase the cost of understanding the original program. We believe that the program protected by our method is quite hard to be understood, and that it is difficult for crackers to cancel the protection, since the dummy instructions are scattered over the program.

We have also implemented a system that automates the construction of self-modifying programs. It can be seen in the experiment that the more we camouflage the instructions, the more expensive the program overhead becomes. That is, the more protected program suffers from the more overhead, which is clearly a trade-off relation.

As seen in the experiment, the self-modification mechanism seems to impose a significant performance overhead compared with the size overhead. 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 (a certain extent of) 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. Specifically, a user of the proposed method should adjust the number of the camouflaged instructions, according to the level of required protection.

Finally, we summarize our future work. We need to investigate the reason why the performance overhead was so expensive for the size overhead. For this, we plan to develop a cleverer algorithm to determine the insertion points of hiding/restoring routines. Also, we conduct experiments with more programs, to find a reasonable application domain of the the proposed method.

References

[1] Albert, D. J. and Morse, S. P., "Combating software piracy by encryption and key management," *IEEE Computer*, pp.68-73, April 1984.

[2] Anazawa, T., "Mobile Music Distribution and its Security Protection," , *IEICE(The Institute of Electronics,Information and Communication Engineers) Office System Workshop*, pp.3-12, May 2001. (in Japanese)

[3] Aucsmith, D. W., "Tamper Resistant Software: An Implementation," In R. J. Anderson ed. *Information Hiding Workshop*, Lecture Notes in Computer Science, Vol. 1174, pp.317-333, 1996.

[4] Aucsmith, D. W. and Graunke, G. L., "Tamper resistant methods and apparatus," *United States Patent*, No. 5,892,899, Assignee: Intel Corporation, Apr. 1999.

[5] Best, R. M. , "Crypto microprocessor for executing enciphered programs," *United States Patent*, No. 4,278,837, July 1981.

[6] Collberg, C., Thomborson, C. and Low, D., "A taxonomy of obfuscating transformations," *Technical Report of Dept. of Computer Science*, U. of Auckland, No.148, New Zealand, 1997.

[7] Collberg, C., Thomborson, C. and Low, D., "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL98)*, San Diego, California, 1998.

[8] Collberg, C., Thomborson, C. and Low, D., "Breaking Abstractions and Unstructuring Data Structures," *IEEE International Conference on Computer Languages(ICCL'98)*, Chicago, IL, May 1998.

[9] Collberg, C. and Thomborson, C., "Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection," *IEEE Transactions on Software Engineering*, Vol.28, No.6, pp.735-746, Aug. 2002.

[10] Drake, C. N., "Computer software authentication, protection, and security system," *United States Patent*, No. 6,006,328, Dec. 1999.

[11] Gailly, J. and Adler, M., "The gzip home page," http://www.gzip.org/.

[12] Hampson, B. E. , "Digital computer system for executing encrypted programs," *United States Patent*, No. 4,847,902, Assignee: Prime Computer, Inc., July 1989.

[13] Hohl, F., "Time limited blackbox security: Protecting mobile agents from malicious hosts," In *G. Vigna ed. Mobile Agents Security*, Lecture Notes in Computer Science, Vol. 1419, pp.92-113, Springer-Verlag, 1998.

[14] Ishima, H., Saitoh, K., Kamei, M., Shin, K., "Tamper Resistant Technology for Software," *Fuji Xerox Technical Report*, No.13, pp.20-28, 2000. (in Japanese)

[15] Kamoshida, A., Matsumoto, T., Inoue, S., “On Constructing Tamper Resistant Software,” *Technical Report of IEICE (The Institute of Electronics,Information and Communication Engineers)*, Vol.97, No.461, pp.69-78, 1997. (in Japanese)

[16] Kanzaki, Y., Monden, A., Nakamura, M., Matsumoto, K., “Protecting Software Programs by Replacing Instructions at Run-time,” *Technical Report of IEICE (The Institute of Electronics,Information and Communication Engineers)*, Vol.102, No.511, pp.13-19, Dec. 2002. (in Japanese)

[17] Mambo, M., Murayama, T. and Okamoto, E., “A tentative approach to constructing tamper-resistant software,” In *Proc. New Security Paradigm Workshop*, Cumbia, UK, 1997.

[18] Monden, A., Takada, Y., Torii, K., “Methods for Scrambling Programs Containing Loops,” *The Transactions of the IEICE (The Institute of Electronics,Information and Communication Engineers)*, Vol.J80-D-I, No.7, pp.644-652, July 1997. (in Japanese)

[19] Monden, A. and Kanzaki, Y., “Program, apparatus and method for adding self-modifying code,” *Japan Patent Pending*, 2002-355881, Dec. 2002. (in Japanese)

[20] Nardone, J. M. , Mangold, R.P., Pfotenhauer, J. L., Shippy, K. L., Aucsmith, D. W. ,Maliszewski, R. L. and Graunke, G. L., “Tamper resistant methods and apparatus,” *United States Patent*, No. 6,178,509, Assignee: Intel Corporation, Jan. 2001.

[21] Ogiso, T., Sakabe, Y., Soshi, M. and Miyaji, A., “Software tamper resistance based on the difficulty of interprocedural analysis,” In *Proc. International Workshop on Information Security Applications (WISA2002)*, pp. 437-452, August 2002.

[22] Paulini, W. and Wessel, D., “Process for securing and for checking the integrity of the secured programs,” *United States Patent*, No. 5,224,160, Assignee: Siemens Nixdorf Informations system AG, June 1993.

[23] Tyma, P. M., “Method for renaming identifiers of a computer program,” *United States Patent*, No. 6,102,966, Assignee: PreEmptive Solutions, Inc., Aug. 2000.

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