A Tentative Approach to Constructing Tamper-Resistant Software

Masahiro MAMBO* Takanori MURAYAMA† Eiji OKAMOTO

School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai Tatsunokuchi Nomi
Ishikawa, 923-1211 Japan

Abstract

So far tamper-resistance has been considered as a property such as information stored *in a device* is hard to read or modify by tampering. Such tamper-resistance is quite important in many situations: superdistribution, electronic commerce systems using IC card, pay television systems with decoders containing secret values for descrambling image and so on. Tamper-resistance ensures proper operation of a program and prevents extraction of secret data and abuse of the program. Moreover, tamper-resistance enables a vendor to enforce his own conditions upon users.

A new notion of tamper-resistance is stated as follows. Tamper-resistance means a property such as information stored *in a device or software* is hard to read or modify by tampering. A tamper-resistant device is usually expensive and not easy to handle compared with its realization in software. It is better to achieve tamper-resistance without relying on any physical device.

Meanwhile, intellectual property rights for a software program can be easily violated once an attacker analyzes the algorithm of a target program. The attacker can create a distinct program which looks quite different but functions just as the target program does. It is very important for software programs to be protected from any reverse engineering and manipulation.

From these observations we study methods to generate a tamper-resistant code and explain an elementary tool, a0/f1/f2/f3. It replaces and shuffles operational codes or inserts reproductive dummy codes in a program so that the output becomes hard to read. After these processes, an attacker conducting algorithm analysis becomes confused by a resultant program possessing unrelevant and irregularly ordered information. Regarding the degree of tamper-resistance, we are still at an elementary stage, but experimental results indicate generated codes have become harder to read.

1997 New Security Paradigms Workshop Langdale, Cumbria UK

1 Introduction

So far tamper-resistance has been considered as a property such as information stored *in a device* is hard to read or modify by tampering. Such tamper-resistance is quite important in many situations. An example is a pay television system. Receivers of the pay television system need to buy a decoder of scrambled image. This decoder includes a secret key inside, which is used directly or indirectly for decoding scrambled image. No company wants illegal receivers to extract and change the secret key and to watch its programs for free. Therefore, the secret key should be protected from observation and modification.

A similar situation occurs in superdistribution [MK90]. The superdistribution is a way of distributing software or digital contents, and a software company charges users for each use of software rather than selling an entire software product. The software company should properly count the use of software and prevent illegal use. The proper operation of the superdistribution system, including enforcement of the conditions set by the software company upon users, is ensured by tamper-resistant devices. Such a tamper-resistant device is stored for instance in user's computer as a coprocessor.

IC card is another example. Some of electronic cash systems using IC cards, e.g. Mondex system, reside their security on the tamper-resistance of IC cards. It is also popular to use IC card for user identification.

*Present Affiliation: Education Center for Information Processing, Tohoku University, Kawauchi Aoba-ku Sendai, 980-8576 Japan, Email: mambo@ecip.tohoku.ac.jp

†Present Affiliation: Konami Enterprise

In either case, embedded secret information should be protected. In general, tamper-resistant IC cards are often used in electronic commerce.

In the Escrowed Encryption Standard (EES) [NIST94] proposed by the American government, an encryption algorithm called Skipjack and other unknown encryption algorithms are stored in a tamper-resistant Clipper Chip. A message in the EES is encrypted by the Skipjack with a session key and the session key is embedded in the Law Enforcement Access Field (LEAF) in an encrypted form. An authorized investigator extracts the session key from the LEAF and conducts a legal wiretapping. The LEAF is computed from the session key and the other values, family key, device unique key and device unique identifier, by utilizing the unknown algorithms (possibly including the Skipjack algorithm). These device unique values are stored in the Clipper Chip. These values, the unknown algorithms and the Skipjack algorithm should not be manipulated by users. Otherwise, the investigator may fail to recover the session key or to decrypt the ciphertext. For successful wiretapping, it is necessary that the LEAF is created by the fixed algorithm and that the device unique values are really related to a clipper chip whose communication the investigator tries to wiretap. A tamper-resistant device achieves the protection of stored algorithms and data. In the similar context, if Trusted Third Party (TTP) is realized by a tamper-resistant device, its proper behavior is ensured by non-manipulation of the device, and user's non-deviation from determined operation is ensured by the difficulty of reading secret values used by the TTP.

Although a tamper-resistant device is quite effective for the protection of algorithms and data, considering cost performance and ease of handling, it is better to achieve the tamper-resistance without using any physical device. Therefore, it is important to broaden the notion of tamper-resistance. In a new notion, tamper-resistance means a property such as information stored *in a device or software* is hard to read or modify by tampering. We discuss techniques of generating tamper-resistant software which protects software algorithms and prevents an attacker from illegal use of software packages [MMOU95, MMOU96, Mura96, MO97, MTT97]. Such secure software is resistant to reverse-engineering and keeps an encryption/decryption algorithm or data secret.

In this paper the concept and the importance of the tamper-resistant code are clarified and an elementary construction for generating such code is shown. Here we focus on assembly level. But that does not mean higher level support is unnecessary for securing software. Work at higher level, i.e. at source level, is also important and will be discussed in the future papers. As another direction of research and development, we can combine software-only approach with hardware-only approach. However, this alternative approach is not discussed in this paper, either.

After the introduction, we discuss effects of tamper-resistant software in Sect.2. Then we show what kind of program is hard or easy to read in Sect.3. Methods used to generate an elementary tamper-resistant code are shown in Sect.4, and their concrete constructions are given in Sect.5. From the experimental results described in Sect.6, we are convinced that a code becomes harder to read than without the tamper-resistant coding operation. Concluding remarks are described in Sect.7.

2 Effects of tamper-resistant software

The tamper-resistant software is expected to be substituted for the tamper-resistant device in most of the situations described in the introduction. If the same level of tamper-resistance is achieved in software as in device, both a maker and a user of tamper-resistant software can get benefit of low cost performance and ease of handling.

One application of the tamper-resistant software is software agent. Whenever a software agent moves to a domain managed by other party, it is under threat of being captured. A captured agent may be scrutinized, and secret information the agent carries may be stolen. Or the agent is modified so that it is abused by the party. The network-type agent should be protected from observation and modification.

In mobile computing a computing device could be lost, and important programs or data would be known by its finder. As long as a program has a tamper-resistant form, it is not revealed even if the computing device is lost or stolen.

In the meantime, our approach has something to do with intellectual property rights. Intellectual property rights of a software program can be compromised once an attacker analyzes the algorithm of a target computer software program. The attacker can create a distinct program which looks quite different but functions just as the target program. He may also create computer viruses, worms or Trojan horses [Ska96] which hack various software tools, software applications and OS programs. Without doubt, these problems prevent the sound growth of a highly computerized society. Software must be protected from reverse engineering. Tamper-resistance of software will provide such protection.

There are several items to be protected in a software package, i.e. the program code, image in the display and the algorithm itself, and different types of legal measures apply to them. In U.S.A., the former two

items are protected by the copyright law and the last one is protected by the Patent Act [Rem82]. On the other hand, it is very difficult in some countries to get a patent for an algorithm used in a software package. The algorithm is not patented without clear relation to the physical property of a physical device. In contrast, a tamper-resistant code protects an algorithm of a program even without any relation to the physical property of a physical device.

Concerning the copyright of a program, it can be protected by combining our approach with an authentication system or a copy-protect system. Once the authentication routine or the copy protection routine is discovered and analyzed, it may be circumvented easily. But if a part of the program for the authentication system or a part of the program for the copy-protect system is made hard to read by a tamper-resistant coding operation, then it becomes harder to find out where such a program part exists in a program. Therefore, an attacker cannot forge a signature or disqualify the protective operation. Since there are not many effective technical measures for protecting software package items, a tamper-resistant code may become a strong candidate for attaining the protection of software packages.

3 Programs that are hard to analyze

3.1 Necessary conditions

In this section we discuss the kinds of program that are hard to analyze. If an algorithm of a program itself is complicated, then there is no doubt that the program cannot be easily analyzed. Then our question is what is the property of a program that possesses a less complicated algorithm and is hard to analyze. In order to proceed our discussion, let us consider what kind of steps we will take when we try to create a program operating similarly to an original program. Obviously, we try to find an algorithm of the target program. Using the given algorithm, we create a modified program with better performance or a distinct program with the same functionality as the original.

The other property of a program that is hard to analyze is that one cannot easily find a module of instructions. For instance, what is our strategy to void the copy-protect mechanism of a software program? No one wants to check the operation of all parts of the program. It is enough to find a module for a copy-protect routine out of the entire program and to simply change a function call for the routine. In this sense, the place of the top and the bottom of a module is important information for the program analysis.

Therefore, a hardly analyzable program satisfies the following property as a necessary condition:

- It is uneconomic to guess an algorithm of a given code. In this category, we include a program with a complicated algorithm.

This property includes a condition such as it is impractical to guess the place of a module in a given code.

At a primary stage of our tamper-resistant code project, we attempt to generate a hardly analyzable program without modifying an original algorithm.

3.2 Properties of an easily analyzable program

An easily analyzable program possesses the following properties:

- The top and the bottom of a module are clearly determined.
- Instructions and registers follow an ordinary rule of a programming, or a distinctive pattern is found in a program.
- The data structure is easily known.

3.3 Properties of a hardly analyzable program

Contrary to the above properties, a hardly analyzable program possesses the following properties:

- The top and the bottom of a module are ambiguous.
- A program code does not have a distinctive pattern.
- The data structure is unknown, and no consistency is found in a data sequence.

4 Methods to generate an elementary tamper-resistant code

4.1 Basic techniques

In order to generate a tamper-resistant code, we aim at converting any instruction pattern useful for program analysis, such as idiom, into unanalyzable instruction patterns without changing an original algorithm. There are several approaches to generating such a tamper-resistant software program. First, instruction streams are irregularly ordered. Second, dummy codes are inserted. Finally, a program is formed by a self-decrypting program. The last approach is closely related to the behavior of polymorphic computer viruses. In this paper,

we mainly discuss how to make instruction streams hard to understand and how to use dummy codes.

The method to make an instruction stream hard to understand consists of two operations. One is a replace operation of specific instructions, and the other is a shuffle operation of instruction stream. The replace operation replaces a complicated operational instruction, such as 'jsr' used for function call, with multiple simple instructions. The shuffle operation shuffles instruction streams. An easily understandable program code has a simple instruction structure which one can intuitively follow. In other words, there are basic rules of programming, and when one reads a code stream, we follow these rules without recognizing them. A complex instruction stream can be separated into several sets of instructions depending on their meaning. These categories are considered to be one of unobservable rules. The replace or shuffle operations make the partition of instruction sets unclear and a produced code does not look like following the rules. Thus the code becomes harder to read.

4.2 Importance of optimization

In addition to the operations described in Sect.4.1, we should optimize redundant instructions. Redundancy in a code helps one to read a code. Redundant instruction optimization removes these redundancies.

In general, a tamper-resistant code generating process should be non-reversible, or if it can be reversed, the reverse operation should be time consuming and expensive. Because otherwise, an attacker can easily recover an original program at a low cost.

Depending on an optimization process and an original program code, the original program becomes either harder or easier to read after the optimization. Even so, an optimized code is composed of simple instructions sets so that it leaves less information to the attacker. It is considered that converting an original program into an optimized code is easy and the the opposite conversion is difficult. In other words, the optimization process is expected to be non-reversible. Hence, we should first optimize an original program in order to get rid of information useful for algorithm analysis. Afterwards, we should execute replacement, shuffling or dummy code insertion.

4.3 Scrambled instruction stream

We use the following rules:

1) To construct a program using only the most fundamental instructions, e.g. add, mov, jcc, jmp, or, and, xor, etc.

2) To optimize codes and, accordingly, to shuffle optimized instructions.

Table 1: Example of conversion

	Original		
	jsr	sub_r	
	jmp	step	
	After replacement		
	add	$-4,sp	;jsr(1)
	mov	$sub_r - 156,-4(sp)	;jsr(2)
	add	$156,-4(sp)	;jsr(3)
	mov	$l - 100,(sp)	;jsr(4)
	add	$100,(sp)	;jsr(5)
	jmp	-4(sp)	;jsr(6)
l:	mov	$step + 135,-4(sp)	;jmp(1)
	add	$-135,-4(sp)	;jmp(2)
	jmp	-4(sp)	;jmp(3)
	After replacement and shuffle		
	add	$-4,sp	;jsr(1)
	mov	$l - 100,(sp)	;jsr(4)
	mov	$step + 135,-8(sp)	;jmp(1)
	add	$100,(sp)	;jsr(5)
	mov	$sub_r - 156,-4(sp)	;jsr(2)
	add	$156,-4(sp)	;jsr(3)
	add	$-135,-8(sp)	;jmp(2)
	jmp	-4(sp)	;jsr(6)
l:	jmp	-12(sp)	;jmp(3)

By these operations, patterns of a program are erased, and the frequency of the use of sources, e.g. the frequency of the use of a register, becomes more uniform. Due to the optimization, the recovery of an original code becomes difficult.

3) Immediate values are not used.

In Table 1, an intuitive example[1] of conversion is shown based on these conversion rules. Since instruction jsr and instruction jmp can be constructed both by jmp, it is not clear whether there is a function call or a branch.

4.4 Dummy codes

There are two types of dummy codes. One is a fresh dummy code shown in Figure 1 and the other is a reproductive dummy code shown in Figure 2. In order to preserve an original operation of a code the effect of inserted fresh dummy codes has to be nullified, e.g. the use of a pair of 'add' and 'sub'. The fresh dummy code does not excessively depend on a target program code. If a generated dummy code is simple, it may be removed

[1] Note that this example is not produced from our program but written by hand. In case of the code sequence with replacement and shuffle, addresses on stuck may be destroyed depending on behavior of a routine program. Stuck pointer should be moved into other place.

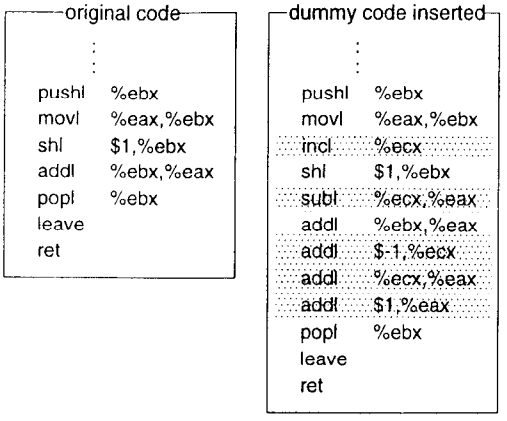

Figure 1: Example of fresh dummy code

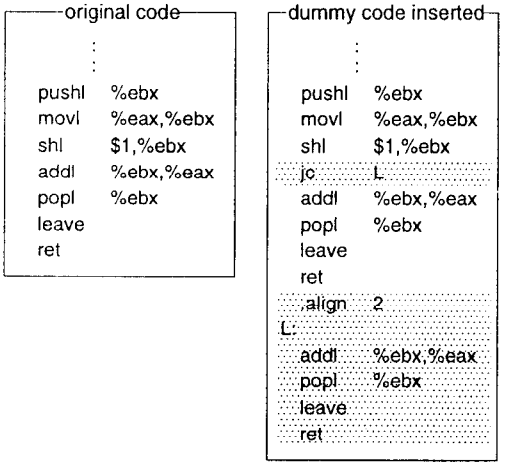

Figure 2: Example of reproductive dummy code

by the optimization. Hence, a fresh dummy code should be complex enough to avoid optimization.

Reproductive dummy code operations generate a dummy code which is a copy of a part of an original code and inserts a conditional branch instruction into the original code. The reproductive dummy code works the same as the original code does. Thus, it is more likely that a program analyst will mistakenly identify the reproductive dummy code as the original code. But if we apply only a reproductive dummy code operation, it can be easily found out by simply observing the generated program code. Therefore, we should apply reproductive dummy code operations together with the replace operation, the shuffle operation and the fresh dummy code operation.

In this paper, we examine only a reproductive dummy code operation among the dummy code operations.

```
According to the occupation of registers,
a shuffling area is determined for each block

    for(i=1;i<No. of lines;i++){
        if(Can i be reordered?){
            Search an area for shuffling → j.
            j → Determine reordered line → j.

            Move line i just before line j.
        }
    }
```

Figure 4: A shuffling algorithm

5 Tools for generation

We show programs a0/f1/f2/f3 which automatically generate an elementary tamper-resistant code of an original gcc assembler code. These programs are created by using yacc, lex and c languages.

- a0, pre-operation, analyzes register information and outputs the register information followed by remarks.
- f1, replace instruction operation, replaces a complicated operational instruction with multiple simple operational instructions.
- f2, shuffle instruction stream operation, shuffles an instruction stream.
- f3, insert dummy instructions, inserts reproductive dummy instructions into a target program.

a0 is a register analysis program and supplies information on free registers in the original code. f1 replaces complicated operational instructions according to information supplied by a0. Several replace operation rules are prepared. f1 applies one among these replacement rules to a target instruction at random. f2 shuffles instruction streams according to information supplied by a0. As shown in Figure 4, f2 decides the shuffle areas in a target program and shuffle patterns at random. For each i, f2 checks whether the line i is moved into the line j. j is determined at random from a region less than the maximum value of j. f3 is an automatic reproductive dummy code insertion tool. First f3 discovers a possible part of the program as the reproductive dummy code. Then it produces and inserts the reproductive dummy code. These procedures are repeated at a predetermined number of times. So the same part is recursively reproduced at maximum by this predetermined number of times. The usage of these programs is as follows.

cat [in] | a0 | f3 | a0 | f1 | a0 | f2 > out

These procedures are shown in Figure 3.

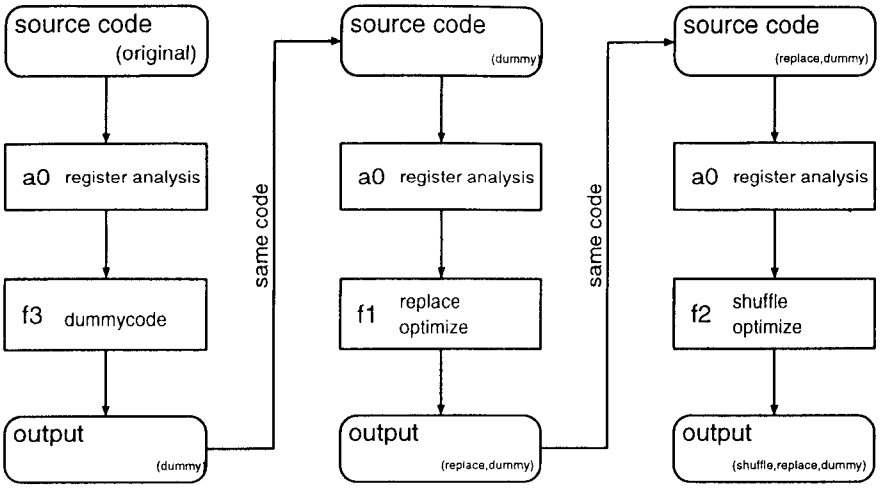

Figure 3: A use of tools

Table 2: Program size and execution time

	Assembler lines	Executable prog. size	Execution time
Original	484 lines	41471 bytes	58.83 sec.
After replacement	510 lines	41471 bytes	58.91 sec.
After replacement*	738 lines	41471 bytes	59.28 sec.

*...with dummy codes

6 Experimental results

6.1 Experiment

In our experiment an encryption/decryption program SAFER.C [Mass93] is compiled by gcc and an output assembler code is processed by the tools explained in Sect.5. A reproductive dummy code is inserted 7 times. The target machine is an Intel 80386.

6.2 Program size and execution time

Table 2 shows the size of a converted assembler file, the size of a obfuscated executable program and the execution time of the executable program. As the execution time, the time of the encryption of a 14.5M byte file is measured.

Without dummy codes, the assembler lines have increased by about 100 lines. This is because a complicated instruction related to function call has been replaced by multiple simple instructions.

With dummy codes, the assembler lines has increased by about 250 lines. On top of the replacement of the complicated instruction, generated reproductive dummy codes make the line longer.

An interesting result is that the size and the execution time of the obfuscated executable program are almost the same as that of the original executable program. These two executable programs are not the same. They actually have 54.6 % different lines between them. We presume the result on the size and the execution time is caused by the following reasons. The most part of the executable program may stem from a link library. In the general compilation process, an object code is generated after assembler process. Then the link library is linked to it and the executable file is created. Our tool manipulates the assembler code and these process do not affect the link library. The reasons for the similar execution time may be that a device access in the SAFER encryption operation takes much of the time, that a simple instruction is faster than a complex instruction and that duplicated redundant codes are removed by the optimization.

6.3 Distribution of opcodes

Fig.5, 6 and 7 show the distribution of opcodes in an assembler program. We can observe the bottom (or the top) of functions at opcodes ret, leave, push and pop before the filtering operation (Fig. 5). But these phenomena disappear after the filtering operation and it becomes hard to find the bottom (or the top) of functions (Fig. 6,7). In the generated code the total number

of used opcodes is reduced and the opcodes mov and add are more frequently used (See also Table 4).

6.4 Code patterns

If a set of consecutive opcodes works as a command, such code pattern could give useful information to a program analyst. It is presumed that the analyst may obtain some information from a long code pattern whether it frequently appears or not. On the other hand, a short code pattern may have less meaning as a program analysis if it frequently appears, and it may have some meaning if it rarely appears.

The number of appearances is accumulated for all code patterns with a certain length and it is shown in Table 3. Before the filtering operation, there are code patterns with the length from 2 up to 31. By the filtering operation, long code patterns have disappeared and the maximum length of code patterns becomes 22 and 23. Both with or without dummy codes, the number of appearances of shorter code patterns have increased and the number of appearances of longer code patterns have decreased. The replacement operation, the shuffle operation and their combination with the optimization operation result in the increase of shorter instructions. On the other hand, because of the reproductive dummy code, for each match length there are more matched patterns in codes with the reproductive dummy code than in codes without it. Hence, it is considered that produced codes give less information than before the filtering operation.

Table 3: Code patterns with a certain length

	Before	After	After*
2 length match	274	283	446
3 length match	186	189	288
4 length match	121	115	188
5 length match	84	77	124
6 length match	57	47	85
7 length match	44	35	60
8 length match	32	23	46
9 length match	24	15	33
10 length match	20	20	26
11 length match	14	11	22
12 length match	13	6	16
13 length match	11	4	13
14 length match	9	4	11
15 length match	8	2	11
16 length match	5	2	7
17 length match	5	2	6
18 length match	4	2	6
19 length match	4	2	4
20 length match	4	2	4
21 length match	2	2	2
22 length match	2	2	2
23 length match	2	-	2
24 length match	2	-	-
25 length match	2	-	-
26 length match	2	-	-
27 length match	2	-	-
28 length match	2	-	-
29 length match	2	-	-
30 length match	2	-	-
31 length match	2	-	-

*...with dummy codes

6.5 Clustering of opcodes

The more uniformly each opcode distributes, the harder the program analysis becomes. As discussed in Sect.6.3, an program analyst cannot find the bottom of a function in uniformly distributed opcodes. Table 4 shows how each opcode are clustered in a program. The number indicates the maximum percentage of each type of opcode as they move through a window of a certain length. This length is set to be 1/10 of the length of the total assembler list. A large value means an opcode are clustered in a small region. We can observe that opcodes are less clustered after the filtering operation. PUSH, POP, LEAVE and RET are removed after the operation. The number of appearance of ADD, MOV and JMP are increased and their percentages are down.

On the other hand, there is no prominent difference between the case with and the case without the reproductive dummy code.

From the experimental results shown in this section, our tentative tools a0/f1/f2/f3 are effective in making a program hard to understand.

Table 4: Clustering of opcodes

	Before		After		After*	
ADD	64 %	(39)	37 %	(67)	32 %	(86)
SUB	60 %	(41)	67 %	(37)	69 %	(42)
DIVL	100 %	(1)	100 %	(1)	100 %	(3)
INC	40 %	(32)	43 %	(32)	47 %	(34)
DEC	52 %	(25)	56 %	(25)	53 %	(32)
AND	100 %	(2)	100 %	(2)	100 %	(2)
OR	100 %	(1)	100 %	(1)	100 %	(1)
XOR	25 %	(28)	25 %	(28)	22 %	(35)
SAX	50 %	(2)	50 %	(2)	50 %	(2)
ROX	100 %	(3)	100 %	(3)	100 %	(3)
LEA	63 %	(11)	63 %	(11)	41 %	(17)
MOV	15 %	(220)	16 %	(254)	14 %	(397)
MOVZX	38 %	(21)	38 %	(21)	36 %	(25)
CMP	45 %	(11)	45 %	(11)	40 %	(15)
JCC	45 %	(11)	45 %	(11)	27 %	(29)
JMP	50 %	(2)	33 %	(6)	26 %	(15)
PUSH	46 %	(15)	-	( )	-	( )
POP	27 %	(11)	-	( )	-	( )
LEAVE	25 %	(4)	-	( )	-	( )
RET	25 %	(4)	-	( )	-	( )

*...with dummy codes
( )... No. of appearance

6.6 Security

We have only shown several evidences of security of our approach. Because of lack of security proof, our approach may be regarded essentially as security through obscurity. It itself is of our interest whether we can prove security of our approach. But even without proof, we can claim the following points. 1) Although most security people usually disapprove of security through obscurity, a reference [Bla96] calls for security based on inherent properties of systems, and really lists the obscurity as one of these inherent properties. 2) Since hardware is expensive and key distribution for cryptography is not easy, almost all copy protection and software license management schemes are based on security through obscurity.

7 Concluding remarks

In this paper we have explained the importance of a tamper-resistant code, and a method to generate such a code has been examined. We have created tools to generate an elementary version of the tamper-resistant code. From the experiment, it has been observed that generated codes become hard to understand. A obfuscated code has increased its assembler line number compared with an original program, but the executable program size and the execution time have not deteriorated.

In terms of copyrights we can combine our method with an authentication system or a copy-protect system. By making authentication programs or copy-protect programs tamper-resistant, authentication or copy-protect operation becomes unavoidable. That is, users have to pass the authentication operation or properly use software programs, which results in enforcement of copyrights. In contrast, the direct use of our method does not always ensure copyrights on a software program. An original program code can be easily converted, or even a converted program code may be converted, into a code with a different appearance, and an illegal user may claim that he has programmed it by himself. It becomes very hard to identify which tamper-resistant code is generated from which original program if the enemy has created the tool from scratch and applies it to non-tamper-resistant software packages. But if software packages have been produced as a tamper-resistant one, or if a user buys the tool for the tamper-resistant code, we should examine further the following protective methods. 1) The tool for the tamper-resistant code should insert hidden identity information into the generated code. This information cannot be easily identified by anyone except the producer of the tool, and it also should be difficult to erase from the generated code. 2) A tamper-resistant code should not be processed by the

tool for the tamper-resistant code again. If we can construct such a tool, a software company does not need to be afraid of further modification of its products once it has generated tamper-resistant software packages.

Another subject we should examine is if it is possible to make maintenance of converted programs. Normally a software company should be able to maintain or update released programs. In the present form of conversion, even an entity who has converted programs cannot easily rebuild original programs, which implies that maintenance seems to be difficult.

On the other hand, it is important to discover evaluation criteria regarding the difficulty of understanding a program. The degree of difficulty of understanding a program may differ in each analyst. In this sense, it may be difficult to estimate the evaluation criteria. But with such criteria one can properly construct a tool for the tamper-resistant code. Moreover, our tool in this paper is at an elementary stage. We need to continue to improve the fundamental techniques used in the present tamper-resistant software.

Acknowledgment

Authors would like to thank NSP'97 participants for their comments. We would also like to thank the Telecommunications Advancement Foundation for the supported.

References

[Bla96] Bob Blakley: *"The Emperor's Old Armor,"* Proc. of 1996 New Security Paradigms.

[Ska96] R. Skardhamar: Virus Detection and Elimination, AP Professional (1996).

[Mass93] J. L. Massey: *"SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm,"* Lecture Notes in Computer Science 809, Fast Software Encryption, Spring-Verlag, pp. 1-17 (1994).

[MTT97] A. Monden, Y. Takada and K. Torii: *"Methods for Scrambling Programs Containing Loops,"* Trans. of IEICE, Vol. J80-D-I, No. 7, pp. 644-652 (July 1997). [In Japanese]

[MK90] R. Mori and M. Kawahara: *"Superdistribution: The Concept and the Architecture,"* Trans. of IEICE, Vol. E73, No. 7, pp. 1133-1146 (1990).

[MMOU95] T. Murayama, M. Mambo, E. Okamoto and T. Uyematsu: *"How to Make a Software Program Hard to Understand,"* Technical Report of IEICE, ISEC95-25, Vol. 95, No. 353 (Nov. 1995). [In Japanese]

[MMOU96] T. Murayama, M. Mambo, E. Okamoto and T. Uyematsu: *"First Step Toward a Tamper-Proof Code,"* The Proceedings of the 1996 Symposium on Cryptography and Information Security, SCIS96-8D (Jan. 1996). [In Japanese]

[Mura96] T. Murayama: *"A Study on Software Protection"* Master Thesis, JAIST (Feb. 1996) [In Japanese]

[MO97] M. Mambo and E. Okamoto: *"Research Topics in Tamper-Resistant Software,"* The Proceedings of the 1997 Symposium on Cryptography and Information Security, SCIS97-10A (Jan. 1997). [In Japanese]

[NIST94] National Institute of Standards and Technology: Escrowed Encryption Standard(EES), Federal Information Processing Standards Publication(FIPS) 185, U.S. Dept. of Commerce (1994).

[Rem82] D. Remer: Legal Care for your Software, NOLO PRESS (1982).

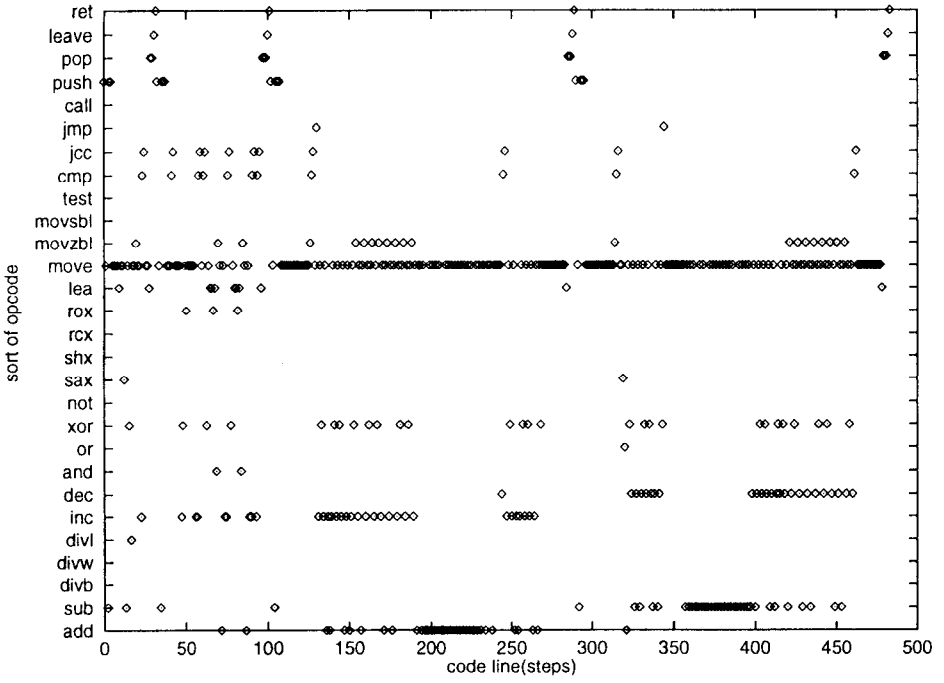

Figure 5: Distribution of opcodes before operation

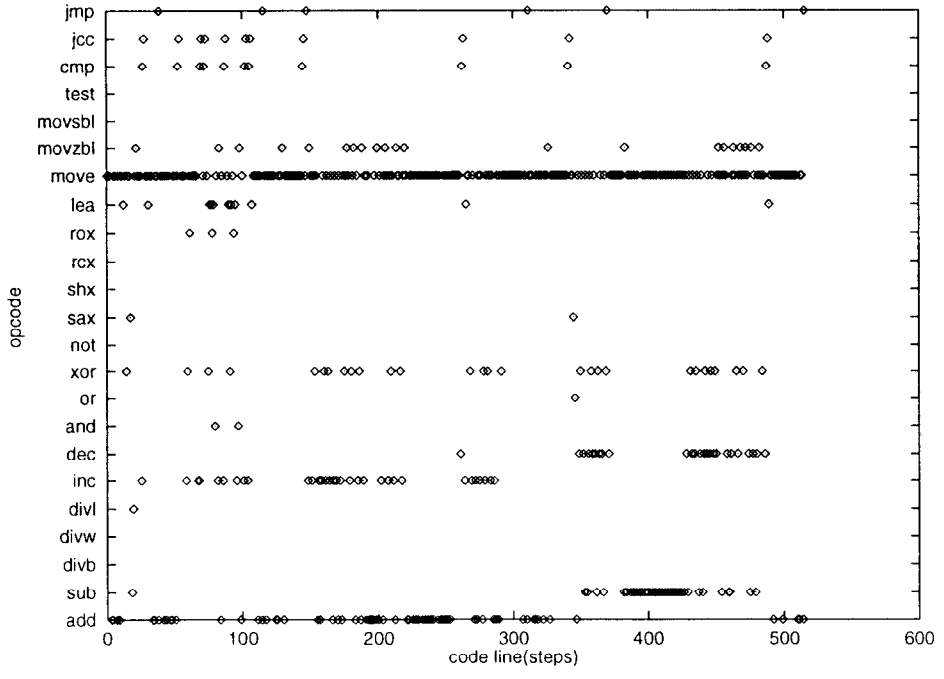

Figure 6: Distribution of opcodes after operation(no dummy code)

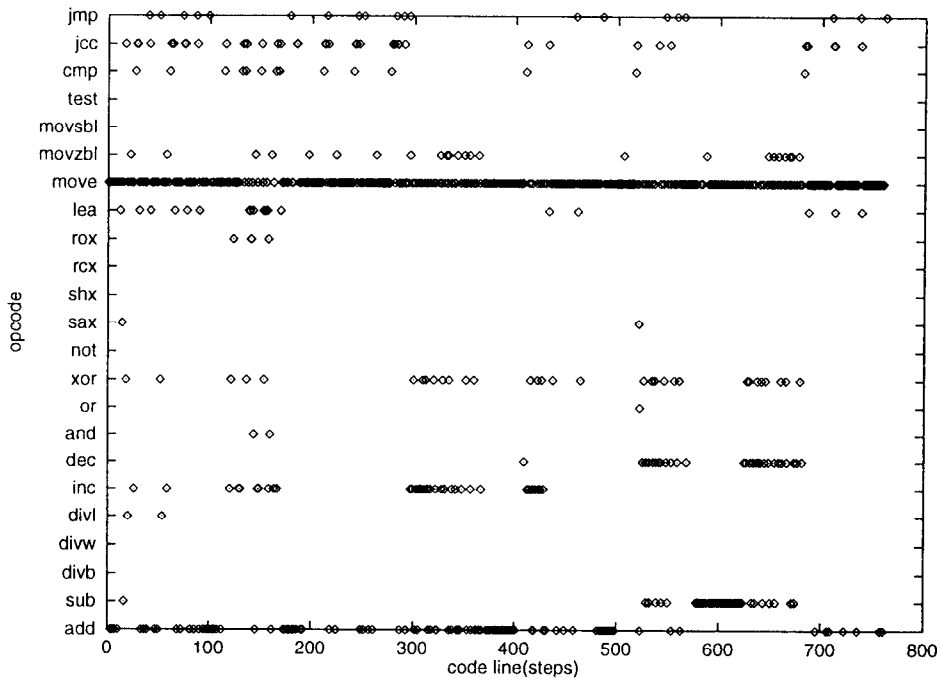

Figure 7: Distribution of opcodes after operation(with dummy codes)