Program Encryption Based on the Execution Time

Hideshi Sakaguchi*, Yuichiro Kanzaki** and Akito Monden***

*Advanced Course of Electronics and Information Systems Engineering,
Kumamoto National College of Technology
Kumamoto, Japan

**Dept. of Human Oriented Information Systems Engineering,
Kumamoto National College of Technology
Kumamoto, Japan

***Graduate School of Information Science,
Nara Institute of Science and Technology
Nara, Japan

***Abstract*— This paper proposes a program encryption method for protecting software against malicious reverse engineering attacks. The code fragments in the program are encrypted beforehand and they are decrypted at runtime using the key derived from the execution time. The proposed method makes the program difficult for adversaries to obtain the secret information by dynamic analysis.**

***Primary categories*— Informatics.**

***Secondary categories*— Software.**

***Keywords*—Software security, software protection, program encryption, program obfuscation.**

I. INTRODUCTION

Many software products contain secret information such as algorithms that are commercially valuable, the secret keys for DRM system, and the routines for license checking. Since such secret information is valuable for malicious users (adversaries), the secrets can be obtained by their reverse engineering attacks, which is a serious threat to software vendors. In order to protect the secret information included in software products against the attacks by the adversaries, software protection methods are required Many software protection methods have been proposed such as program obfuscation, program encryption, and software tamper-proofing techniques [4]. The basic idea of the program encryption (e.g. [1] and [2]) is to encrypt the code fragments in the program before execution and decrypt/re-encrypt them at runtime. It is effective to make the program difficult to analyze the code fragments by static analysis because the program encryption transforms them into meaningless code. However, the program encryption is not always effective in complicating dynamic analysis, since the adversary can stop the execution of the program using a debugger at the time the encrypted code is decrypted and obtain the original code.

We propose a program encryption method which aims to especially complicate dynamic analysis. In our method, the key is generated from the execution time taken to execute a designated block of the program. If the time of the block is within the predetermined range (the execution time is valid), the encrypted code becomes the original one. However, if the execution time is out of the predetermined range (the execution time is invalid) due to dynamic analysis, the encrypted code becomes the different one.

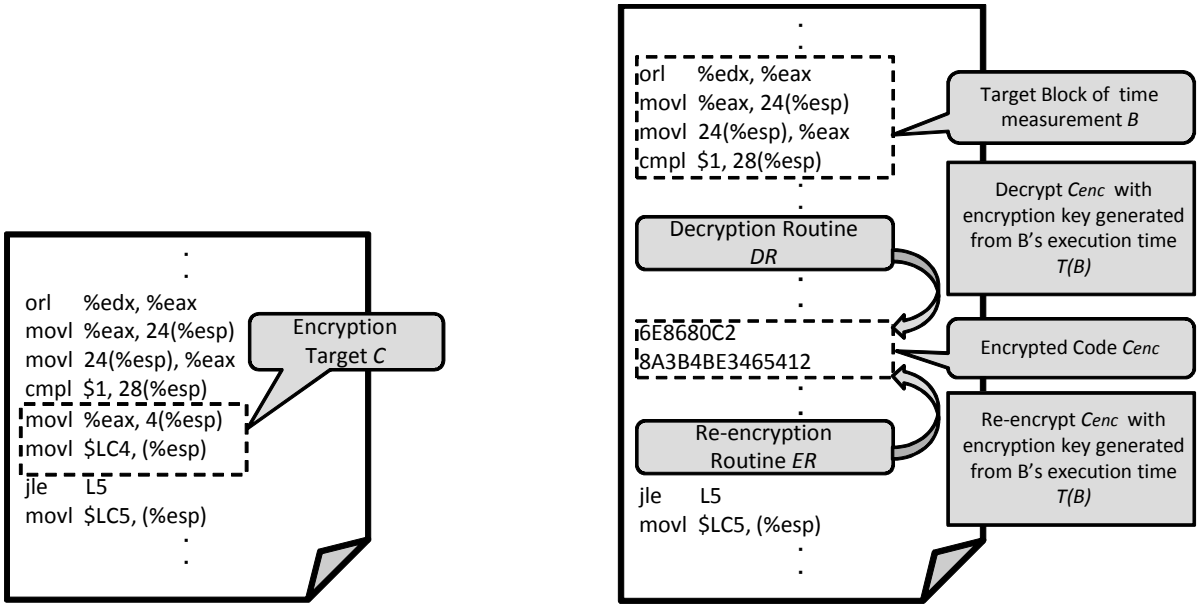
The rest of this paper is organized as follows: In Section II, we show the basic idea of the method. In Section III, we explain the procedure of applying the method to programs. In Section IV, we conduct a case study to examine whether a protected program is effective against dynamic analysis. In Section V, we describe the current problems of the proposed method. In Section VI, we review the related work. In Section VII, we conclude the paper.

II. BASIC IDEA

First, we show the basic idea of our method. Our method aims to protect a program by adding many routines that correctly decrypts encrypted code only if the execution time is valid. Fig.1 (a) and (b) show the examples of the original program P and the protected program P_p, respectively. The examples are shown by AT&T assembly code. C is the encryption target, B is the target block of time measurement, $T(B)$ is the execution time of B, DR is the decryption routine, ER is the encryption routine, and C_{enc} is the code which is generated encrypting C by symmetric key encryption scheme. A part of P is selected as C and C is overwritten with C_{enc}. C_{enc} is decrypted by DR and C_{enc} is re-encrypted by ER at runtime, i.e., C appears only during the time between when C_{enc} is decrypted and when C_{enc} is re-encrypted. The key which is used when decrypt/re-encrypt C_{enc} is generated from $T(B)$. The proposed method is executed as follows:

1. When the execution reaches B, the execution time $T(B)$ is measured. $T(B)$ is hashed to $hash(T(B))$ by one-way hash function and $hash(T(B))$ is stored in the memory.
2. When the execution reaches DR, C_{enc} is decrypted with $hash(T(B))$ as the key.
3. When the execution reaches C_{enc}, the decrypted C_{enc} (same code as C) is executed.
4. When the execution reaches ER, C_{enc} is encrypted again with $hash(T(B))$ as the key.

If $T(B)$ is longer or shorter than $T_0(B)$ (valid execution time of B), $hash(T(B))$ does not match $hash(T_0(B))$. It means to $hash(T(B))$ is the invalid key (the valid key is $hash(T_0(B_i))$) and C_{enc} is not decrypted into C when DR is executed. If C_{enc} is not



(a) Original program P (b) Protected program P_p

Figure1. Basic idea

decrypted into C, the decrypted code does not perform the correct behavior. Our method restricts the valid execution time range in order to detect reverse engineering attacks. If the adversary executes P_p under debugger, the execution time becomes invalid and the valid key is not generated. Therefore, we see our method is effective against dynamic analysis especially.

III. Procedure For Applying Our Method

A protected program P_p is obtained by repeating the following six steps. We assume that the following steps are in the i-th iteration of the process. The i-th C, C_{enc}, B, DR, and ER are denoted as C_i, C_{enc_i}, B_i, DR_i, and ER_i, respectively.

(Step 1) Determining the encryption target C_i

At first, we determine the encryption target C_i to be encrypted. We select a code fragment of the original program P as C_i. We usually select a secret part of the program such as conditional branch, a key used for encryption/decryption of digital contents, and a valuable algorithm as C_i. C_i is transformed into the encrypted code (called C_{enc_i}) and C_i is overwritten with C_{enc_i} in (Step 6).

(Step 2) Determining the Target Block of Time Measurement B_i and the positions of the routines DR_i and ER_i

We select the target block of time measurement B_i and the positions of inserting the decryption routine DR_i and the encryption routine ER_i. They are determined that they will satisfy the following conditions:

1. B_i is a basic block that exists on the path from the entry point of P_p to C_i.
2. DR_i is inserted at a point between B_i and C_{enc_i}.
3. ER_i is inserted at a point between C_{enc_i} and the end of P_p.

(Step 3) Inserting instructions for measuring the time of B_i

We insert time measurement instructions just before B_i and just after B_i. $T(B_i)$ means the execution time taken to execute B_i. We can measure $T(B_i)$ using the instruction which reads the time stamp counter (such as RDTSC instruction in the Intel A-32 architecture [3]). We then insert one-way hash function after the time measurement of B_i. The hash function generates $hash(T(B_i))$, the hash value of $T(B_i)$.

(Step 4) Generating Decryption Routine DR_i and Encryption Routine ER_i

We generate the decryption routine DR_i and the encryption routine ER_i. DR_i is to restore C_{enc_i} to the original code C_i at runtime using $hash(T(B_i))$ as the key. ER_i is to encrypt the C_i to C_{enc_i} again using $hash(T(B_i))$ as the key. DR_i and ER_i are inserted into the positions determined in (Step 2).

(Step 5) Determining the threshold time

We determine the threshold time on the assumption that the execution time becomes longer or shorter if the adversary executes P_p under a dynamic analysis tool (e.g., a debugger). We determine $T0_{min}(B_i)$, the minimum execution time of B_i under normal execution, and $T0_{max}(B_i)$, the maximum execution time of B_i under normal execution. If $T(B_i)$ falls between $T0_{min}(B_i)$ and $T0_{max}(B_i)$, we judge normal execution is operating and $T(B_i)$ is the valid execution time. We determine them in advance by executing B_i or estimate the approximate execution time according to the code that constructs B_i and the execution environment.

(Step 6) Overwriting C_i with C_{enc_i}

We overwrite C_i with C_{enc_i}. We use $hash(T_0(B_i))$, the hash value of the valid execution time of B_i, as the key of the encryption. We determine $T_0(B_i)$ according to $T0_{min}$ and $T0_{max}$. We overwrite C_{enc_i} on C_i after we encrypt C_i into C_{enc_i}.

IV. CASE STUDY

In this section, we examine the behavior of a program protected by our method. In this case study, the protected program has routines for checking serial number and expiration date. Fig. 2 (a) and (b) show the flow of the original program *P* and the flow of the protected program P_p, respectively. This time, we select the routine for checking expiration date as the encryption target *C*, and the routine for checking serial number as the target block of time measurement *B*. Additionally, we employ 128-bit AES in ECB mode as the symmetric key encryption scheme and use MD5 as the one-way hash function. Table I shows the execution environment. We measure the execution time in clock cycles. $T0_{min}(B)$ is set to 1,048,576 clock cycles and $T0_{max}(B)$ is set to 2,097,151 clock cycles.

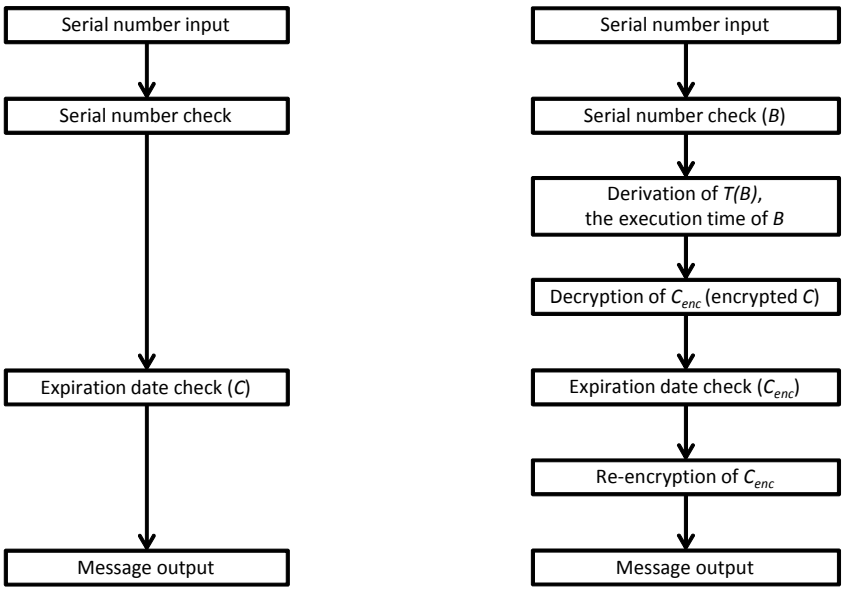
We execute P_p in five different manners as follows:

- Normal execution.
- The execution of the program is paused at *B* for approximate three seconds using the breakpoint function of the debugger.
- All of the executed instruction in *B* are written to the file.
- Instruction in a part of B (approximately 10% of instructions in *B*) are written to the file.
- Instruction in a part of B (approximately 10% of instructions in *B*) are skipped.

The results of each execution are shown in Table II. In the 'Result' column in table II, the words "correct" and "wrong" mean that C_{enc} is decrypted into the original code, and C_{enc} is not decrypted into the original code, respectively. C_{enc} was correctly decrypted only when P_p was normally executed. On the other hand, C_{enc} was not correctly decrypted when the

TABLE I. Execution environment

OS	Windows 7 Home Premium 64-bit
CPU	Intel(R) Core(TM) i7 CPU @ 2.80GHz
Memory	4.00GB



(a) Original program *P* (b) Protected Program P_p

Figure 2. Execution flow

TABLE II. Execution results

Execution manner	Execution time [clock cycles]	Proportion of the execution time to the normal execution time	Result
Normal execution	1,369,098	1.00 times	Correct
The execution is paused at *B* for three seconds	9,679,263,929	Approx. 7,070 times	Wrong
All of the executed instruction of *B* are written to the file	49,434,046	Approx. 36.1 times	Wrong
All of the executed instruction of *B* are written	28,688,669	Approx. 21.0 times	Wrong

to the file			
All of the executed instruction of *B* are skipped	944	Approx. 6.90×10^{-4} times	Wrong

execution of the program was paused at *B*, the executed instruction of *B* (both part and all) were written to the file, and instruction in a part of *B* were skipped. When C_{enc} is not decrypted into *C* correctly, exceptions (illegal instruction, access violation, and privileged instruction) occur. In terms of the execution time, it took approximate 7,070 times in case of the execution of the program is paused at *B* for three seconds, approximate 36.1 times in case of all of the executed instruction in *B* are written to the file, approximate 21.0 times in case of instruction in a part of *B* are written to the file, and approximate 6.90×10^{-4} times in case of instruction in a part of *B* are skipped, respectively compare to the execution time in case of normal execution.

As seen in this experiment, C_{enc} is decrypted into the original code correctly when P_p is normally executed. On the other hand, if the execution time of a certain part of the program is changed due to dynamic analysis, C_{enc} is overwritten with uncertain code.

V. DISCUSSION

We have proposed a program encryption method which aims to protect against dynamic analysis. In the case study described in Section IV, we showed that the method is effective against dynamic analysis in certain circumstances. Then, we suggest below things to improve the method.

First, we make the target block of time measurement *B* more difficult to find. If the adversary has knowledge about our method, he could obtain the original code *C*. He could obtain *C* by finding *B* and normally execute *B*. In our method, *B* is put between the time measurement instructions. He could find *B* from the positions of the time measurement instructions. Then we suggest protecting the time measurement instruction by our method and we suggest inserting the dummy time measurement instructions in many positions of the protected program.

Secondly, we adjust the threshold time to the protected program runs under the different execution environments. In practical situation, the protected program would be executed under the various execution environments. Therefore the execution time changes in each of execution environment. Then it is required to adjust the threshold time $T0_{min}(B)$ and $T0_{max}(B)$ according to the execution environment.

Thirdly, we reduce the performance overhead of the protected program P_p. The execution time of P_p is longer than the original program *P* due to the inserted routines and instructions. The execution time of P_p which is used in the case study is approximate 5.81 times longer than the one of *P*. Then we suggest deploying fast algorithm for hashing and encrypting/decrypting.

VI. RELATED WORK

There have been methods for encrypting program. For example, Cappaert et al. proposed a program encryption method [2]. In the method, all functions (except for the main function) are encrypted beforehand. Each function is decrypted just before the caller of the function jumps to the function and the function is re-encrypted just after returning to the caller of the function. Aucsmith et al. proposed another program encryption method [1]. In the method, a function is split into pieces (called cells) and the cells are separated into two groups. Each cell of a group is xored with the cells of another group and is encrypted. The method continuously takes xor and encryption round during execution. Each cell is transformed into the cleartext before the cell is executed. Our method is different from the above methods in that the execution time is exploited for protecting against dynamic analysis.

There also have been software protection methods based on the execution time. For example, Kanzaki et al. proposed a program camouflage method [5]. The instructions which are camouflaged with other instructions are restored according to the execution time at runtime. Our method is different from this method in that the encryption technique is used to transform the code. Collberg et al. also proposed the software protection method [4]. In the method, the execution time is compared with the threshold time at the conditional branch. The instruction that is executed is determined according to the result of the comparison. Our method is different from the method in that the execution time is used to generate the key.

VII. CONCLUSION

In this paper, we proposed a program encryption method which aims to protect against dynamic analysis. The code fragments in the program are encrypted with the symmetric key encryption scheme beforehand. The encrypted code fragments are decrypted/re-encrypted at runtime. The key is generated from the execution time. We examined the behavior of the program protected by our method in Section IV. It was shown that the cleartext of the encryption target does not appear when the execution time is invalid due to dynamic analysis.

A foreseeable extension of our method would be to make the inserted codes such as time measurement instruction and encryption routine difficult to analyze against the adversary, adjust the threshold time for different execution environments, and reduce the performance overhead of the protected program.

REFERENCES

[1] D. W. Aucsmith. Tamper Resistant Software: An Implementation. In *Information Hiding*, Vol. 1174 of *Lecture Notes in Computer Science*, pp. 317-333. Springer-Verlag, 1996.

[2] J. Cappaert, N. Kisserli, D. Schellekens and B. Preneel. Self-encrypting Code to Protect Against Analysis and Tampering. *Proc. Benelux Workshop on Information and System Security*, 2006

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

[4] C. Collberg and J. Nagra. *Surreptitious Software*. Addison-Wesley, 2009

[5] Y. Kanzaki and A.Monden. A Method for Hiding Program Code Focused on the Execution Time Difference, Vol. 1 of *Lecture Notes in FIT 2009*, pp. 361-364, September 2009.