Secure and Efficient Code Encryption Scheme Based on Indexed Table

Sungkyu Cho, Donghwi Shin, Heasuk Jo, Donghyun Choi, Dongho Won, and Seungjoo Kim

Software is completely exposed to an attacker after it is distributed because reverse engineering is widely known. To protect software, techniques against reverse engineering are necessary. A code encryption scheme is one of the techniques. A code encryption scheme encrypts the binary executable code. Key management is the most important part of the code encryption scheme. However, previous schemes had problems with key management. In an effort to solve these problems in this paper, we survey the previous code encryption schemes and then propose a new code encryption scheme based on an indexed table. Our scheme provides secure and efficient key management for code encryption.

Keywords: Code encryption, reverse engineering, software protection, tamper resistance.

I. Introduction

The Business Software Alliance estimated that the monetary value of copyright infringement was US $53 billion in 2008 [1]. This was up US $5.1 billion from 2007. The major reason for copyright infringement is that software is completely exposed to an attacker after it is distributed since reverse engineering is widely known [2]-[5].

Therefore, to protect software, techniques against reverse engineering are necessary. A code encryption scheme is one of the techniques. A code encryption scheme encrypts the binary executable code. This is accomplished by encrypting the program at some point after it is compiled [6], [7]. However, skillful reverse engineers can easily find the secret key. To solve this problem, a code encryption scheme needs secure key management.

Cappaert [8], [9] and Jung [10] have proposed code encryption schemes that generate a secret key with the related information of a binary code at runtime. However, previous schemes have problems with key management. First, Cappaert's scheme cannot generate the correct secret key. If a secret key is not generated properly in a code encryption scheme, it may lead to program crashes or other unintended behavior. Second, the size of an executable file is increased considerably in Jung's scheme; this may lead to an efficiency problem.

To solve the problems, we review the previously proposed code encryption schemes, and then we propose a new code encryption scheme based on an indexed table included in this paper. Our scheme generates the correct key in any software which has various structures and with performance advantage.

The remainder of this paper is structured as follows. In section II, we describe the existing code encryption schemes. In

Manuscript received Jan. 26, 2010; revised June 28, 2010; accepted July 15, 2010.

This research was supported by the Ministry of Knowledge Economy (MKE), Rep. of Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA)" (NIPA-2010-(C1090-1031-0005)), and also supported by the IT R&D program of MKE, Rep. of Korea [Development of Privacy Enhancing Cryptography on Ubiquitous Computing Environment].

Sungkyu Cho (phone: +82 10 4301 3350, email: skcho@security.re.kr), Donghwi Shin (email: dhshin@security.re.kr), Heasuk Jo (email: hsjo@security.re.kr), Donghyun Choi (email: dhchoi@security.re.kr), Dongho Won (email: dhwon@security.re.kr), and Seungjoo Kim (skim@security.re.kr) are with the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Rep. of Korea.

doi:10.4218/etrij.11.0110.0056

section III, we propose our encryption scheme based on an indexed table. We compare previous schemes in section IV. Finally, our conclusions and a suggestion of possible future work are in section V.

II. Related work

Code obfuscation and encryption are used to protect software [11]-[13]. However, code obfuscation merely makes it time-consuming, but not impossible, to reverse a program. Therefore, we concentrated on code encryption which can protect the software from reverse engineering. We searched for schemes that are related to code encryption and found two applicable schemes. In this section, we briefly review both Cappaert's and Jung's schemes.

1. Cappaert's Scheme

Cappaert proposed a partial encryption scheme based on a code encryption scheme [8], [9]. To apply the partial encryption scheme, binary codes are divided into small parts and encrypted. The encrypted binary codes are decrypted at runtime by users. In this way, the partial encryption overcomes the weakness of revealing all of the binary code at once because only the necessary parts of the code are decrypted at runtime. Cappaert's scheme is shown in Fig 1.

As shown in Fig. 1, the scheme relies on function encryption and code dependency. For example, if a *calc* function invokes a *sum* function, the secret key which is used to encrypt a *sum* function is the *calc* function's own binary code. The *sum* function is decrypted at runtime, and then the *calc* function is

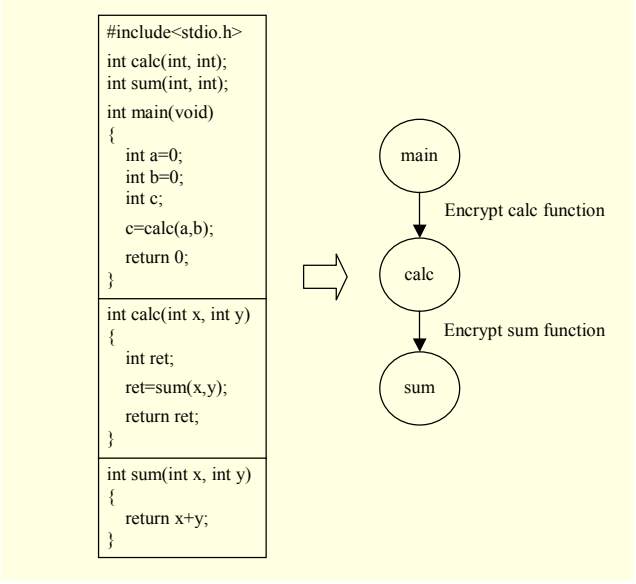


Fig. 1. Example of Cappaert's code encryption scheme.

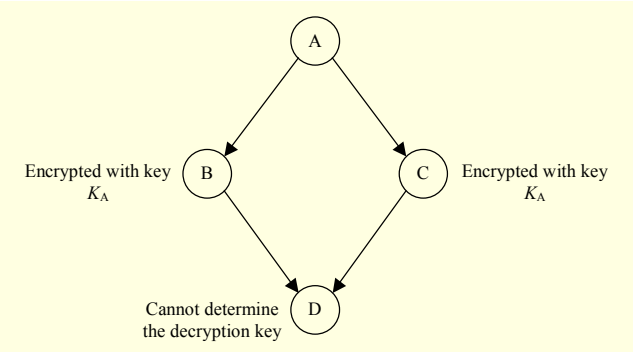


Fig. 2. Problem of key generation.

decrypted, which invokes a *sum* function. When the *sum* function completes the work, it is encrypted again and stored in the memory.

In this scheme, all functions decrypt or encrypt another function using their own code as a key material. This ensures protection against tampering. If an attacker attempts to tamper with the protected program execution, the program outputs an incorrect binary code. Consequently, the binary code will cause incorrect execution and undesired behavior.

Cappaert's scheme protects its information using code encryption, but it does not perform correctly when a function is invoked by multiple functions. We assume that there are functions (*A*, *B*, *C*, and *D)* in the program, as shown in Fig. 2.

According to the scheme, the secret key of function *D* should be determined from *B* or *C*, but Cappaert's scheme cannot determine which secret key is used. Cappaert's scheme has no solution to the problem of determining the secret key. For this reason, Cappaert's scheme can only be applied to software that has a single path and cannot be applied to generic software that has multiple paths.

Cappaert proposed an improved scheme two years later [8]. It is similar to the original scheme, but it has a few modifications. The original scheme, which was proposed in 2006, uses the caller's own binary code as the secret key, but the improved scheme, which was proposed in 2008, used the caller's own hash value of the binary code as the secret key. This can provide tamper-resistance, but key generation and management problems are still present.

2. Jung's Scheme

Jung and others proposed a code block encryption scheme to protect software using a key chain. Jung's scheme uses a unit block, that is, a fixed-size block, rather than a basic block, which is a variable-size block. Basic blocks refer to the parts of codes that are divided by control transformation operations, such as "*jump*" and "*branch*" commands, in assembly code [8], [9].

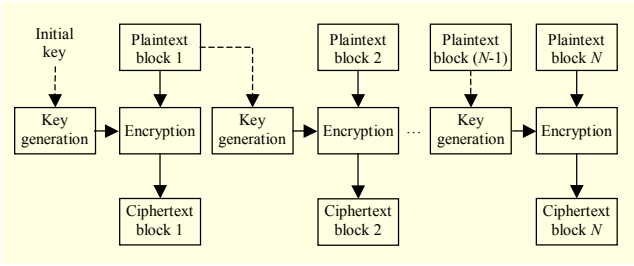


Fig. 3. Jung's code encryption scheme.

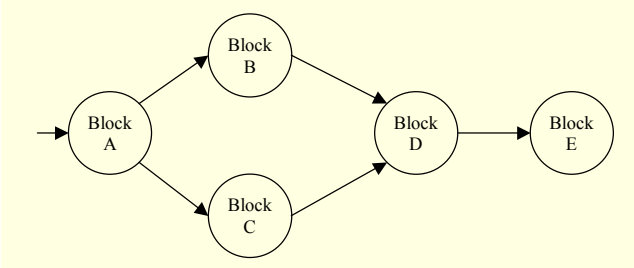


Fig. 4. Flow of the example program.

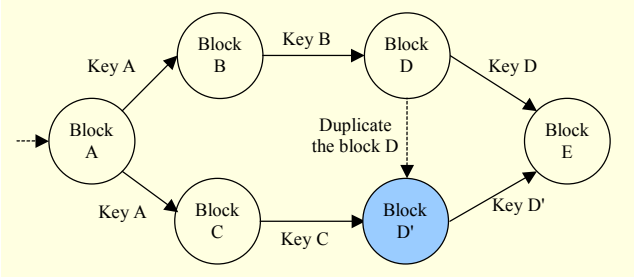


Fig. 5. Key chain in Jung's scheme [10].

Jung's scheme is very similar to Cappaert's scheme. As shown in Fig. 3, this scheme encrypts the *N*-th block using the key that was created from the (*N*–1)th block.

Jung's scheme tries to solve the problem of Cappaert's scheme. If a block is invoked by more than two preceding blocks, the invoked block is duplicated. We assume the flow of an example program as shown in Fig. 4.

As shown in Fig. 4, the secret key of block *D*, which is invoked by multiple blocks, should be chosen as block *B* or block *C*. According to the Jung scheme, a key chain is constructed as shown in Fig 5.

At this time, by duplicating the block *D* on another path, *D*' is created. The secret key of *E* is determined by block *D* or block *D*'; block *D* and block *D*' are identical. The secret key of block *D* is determined by block *B*, and the secret key of block *D*' is determined by block *C*. Therefore, the secret key of *E* is determined correctly even though the block is invoked by more than two blocks. A key chain can be achieved in this way, and then the encrypted code is stored as an executable file.

Jung's scheme solves the problem of determining the secret key of a block which is invoked by more than two preceding blocks. However, it has a disadvantage in aspect of the code size. According to the scheme, the executable file size is increased not only in procedure of converting a basic block to a unit block, but also in the duplicating procedure. In addition, if the numbers of preceding blocks are increased, the number of duplicated blocks is also increased equivalently.

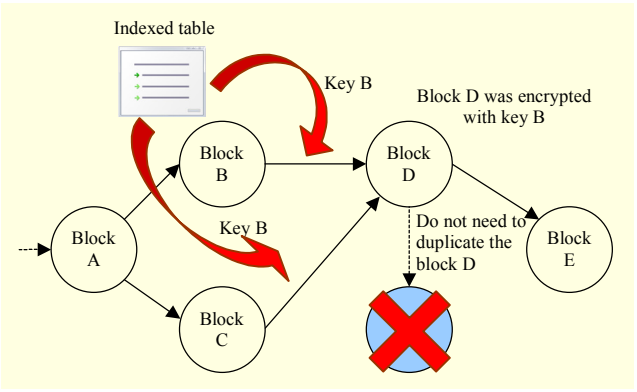


Fig. 6. Novelty of our scheme.

To solve this problem, we propose a new scheme based on an indexed table in this paper. In our scheme (assume that the flow of an example program is as shown in Fig. 4), block *B* and *C* can decrypt block *D* without duplicating block *D* since block *B* and *C* can acquire the key *B* through the indexed table, as shown in Fig. 6.

III. Proposed Scheme

We propose a code encryption scheme based on an indexed table to protect software. The indexed table can solve the problem of multiple paths. In addition, it solves such problems as loops, recursions, and multiple calls.

1. Notations and Requirements

The notations in Table 1 are used throughout this paper. *IK* is the initial key that protects both the indexed table and the random number. The random number encrypts basic blocks that are invoked by multiple blocks, and the *IK* encrypts the random number. Providing and managing the *IK* depends on the application, and the *IK* can be stored in an external devices, such as an external hard drive or a trusted platform module. Hence, we assume that *IK* was distributed offline and stored securely. A random number is used to encrypt a multiple called block, and the random number is encrypted with *IK*. *PK* denotes the encrypted random number using the key *IK*, that is, $PK=Enc_{IK}(r)$. The *PK* is stored in the data section of executable binary code. Symmetric encryption and decryption with secret key *K* are denoted by $Enc_K(\cdot)$ and $Dec_K(\cdot)$,

Table 1. Notation.

Notation	Description
IK	Initial key
PK	Protected key
$Enc_K(\cdot)$	Symmetric encryption with key K
$Dec_K(\cdot)$	Symmetric decryption with key K
$H(\cdot)$	One-way hash function
r	Random number
A to Z	Basic blocks in binary code
P_i	Basic block of program in plain form
C_i	Basic block of program in encrypted form

respectively. However, we will not discuss the encryption or decryption algorithm itself because it is outside of the scope of this paper.

A few requirements are necessary for the secure code encryption scheme:

• **Confidentiality.** The original binary code should be protected from static analysis by maintaining confidentiality. To protect a binary code from a dynamic analysis, which analyzes data flow and control transformation, a minimal number of code blocks should be present in the memory. As long as the code remains encrypted in memory, the program can be protected from static and dynamic analysis [8].

• **Memory dump prevention.** If a single routine encrypts a whole program, the decryption routine decrypts the body and sets up the starting point of the body as the entry point. At this time, it can be easily cracked by a memory dump. Therefore, only a small part of the program should be decrypted, while the other parts of the program remain in encrypted form.

• **Correct key chain.** When a code encryption is applied to a program, the correct key chain is required. If it does not have a correct key chain for the multiple paths, the system can crash or engage in an undesired execution.

• **Tamper resistance.** To protect from tampering, and maintain integrity [8], [15], [16], we want the following properties:

- In the encryption process, a one-bit change in a basic block A affects all following ciphertext blocks.
- In the decryption process, if one or more bits are modified in encrypted basic block B', the result of decryption should be changed by one or more bits.

2. Code Encryption

All of the steps of the code encryption algorithm are shown

```
Procedure encryption()
 1. Compile();   // compile the source code to generate object or
                    executable file
 2.
 3. entrypoint ← Find_EntryPoint();   // store an address of
                                         entry point
 4. currentAddress = entrypoint;      // initialization
 5. nextAddress = 0;
 6.
 7. ConstructTable()   // this procedure is described in Fig. 10.
 8.
 9. nextAddress = Find_next(entrypoint);   // find an address of
                                         next block in current block
10. Encrypt(IK, entrypoint);   // encrypt first block
11.
12. while(File pointer is not end of file)
13. {
14.       if(SearchTable(nextAddress)) // if next block's flag in
                                         the indexed table indicates 1
15.       {
16.             random = GenerateRandom(); // generate random
                                                number
17.             Encrypt(random, nextAddress); // Encrypt with the
                                                random number
18.             Encrypt(random, IK);   // Encrypt the random
                                         number with the IK
19.       }
20.       else                          // next block's flag indicates 0
21.           Encrypt(random, currentAddress);   // Encrypt with
                                                the current block
22. }
```

Fig. 7. Pseudo-code to encrypt executable file.

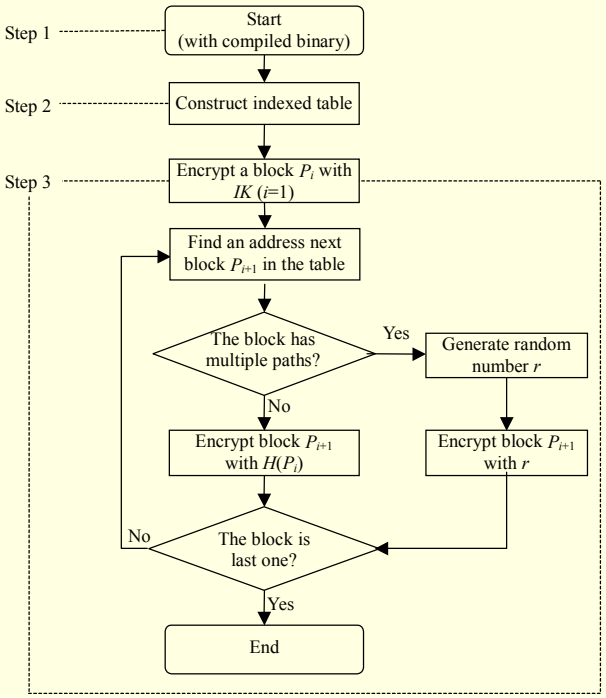


Fig. 8. Encryption process flow chart.

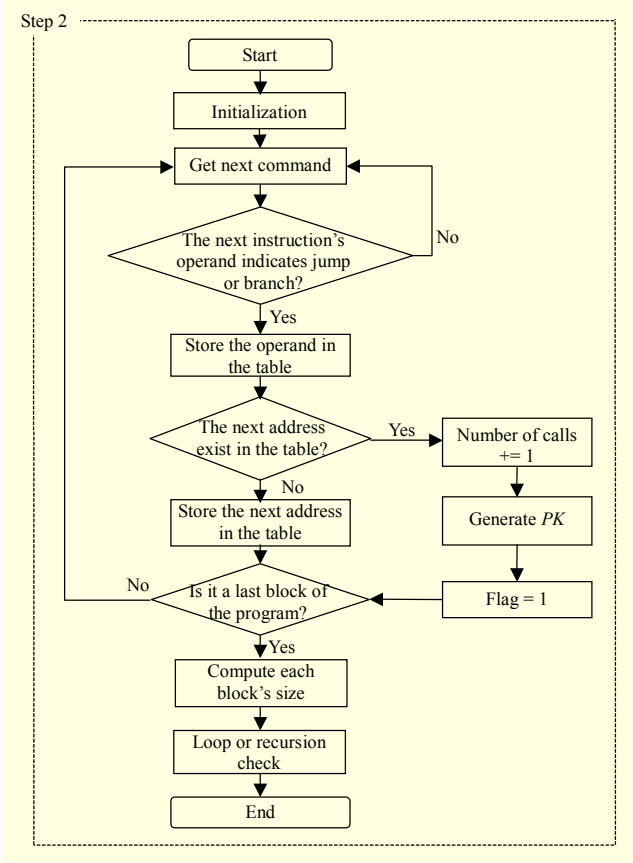


Fig. 9. Detailed flow chart of step 2.

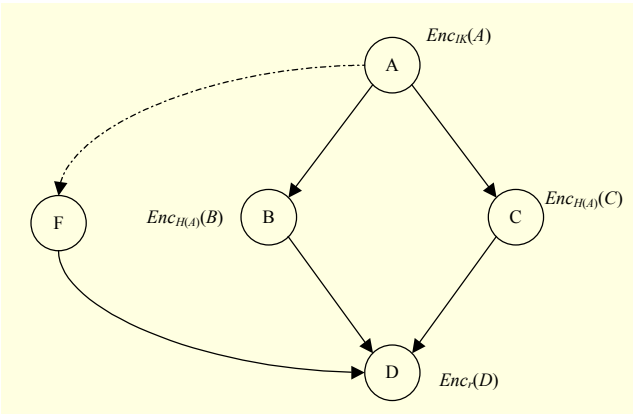


Fig. 10. Key chain in our scheme.

in Fig. 7. These code encryption steps are described in the flow chart of Fig. 8.

Step 1 is the source code compiling process. After this step, the source code is compiled and outputs a binary image.

Step 2 is construction of the indexed table. It is the most important procedure of our scheme. We describe the details of step 2 in Fig. 9.

Lastly, in step 3, the basic blocks are encrypted. This is $C_i = Enc_{H(P_{i-1})}(P_i)$, where $H(P_0) = IK$. At this time, i is

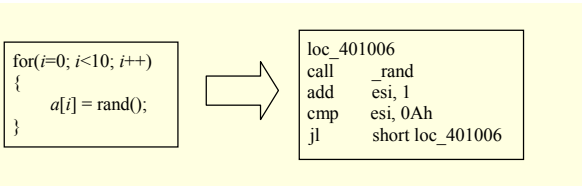


Fig. 11. C code and its assembly code.

the number of blocks in the indexed table. If a flag is not zero, the block is encrypted by a random number. *IK* encrypts the random number which is then stored in an executable image. This encryption process is needed because if the random number is exposed, the blocks can be decrypted by the random number and analyzed by the attacker.

The indexed table is used to make the correct key chain. The table construction procedure is as follows. First, store the current address of the basic block, and investigate the "*jump*" or "*branch*" command in the basic block by moving the pointer. The commands contain a block's address, which will be executed next. If the next address refers to the current address of the basic block, this indicates a loop or recursion. When a loop or a recursion occurs due to the "*cmp*" command with the number of calls, the number of calls is marked in the table. Likewise, if a current address of a block is stored in the table already, this indicates multiple calls. The *PK* is generated at this time and stored in the binary image in a data section.

As shown in Fig. 10, a basic block *D* is invoked by multiple blocks *B*, *C*, and *F*. The secret key of *B* is the hash value of block A, and the secret key of block *C* is the same. *F* is not invoked by block *A* directly, but it invokes the basic block *D*. At this time, random number *r* is generated for the secret key of *D*, and then it is encrypted with *IK*. The result of the encryption is a *PK*. The *PK* is stored in executable images. Generic operating systems, such as Windows or Linux, store variables in the data section of an executable image. Therefore, the *PK* is stored in the data section of an executable image.

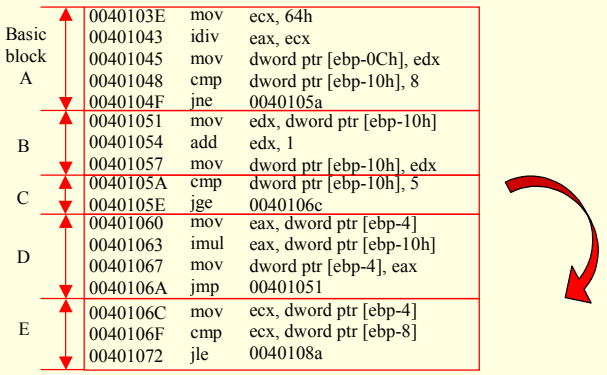
The indexed table contains the number of iterations and recursions. If this is not considered, a basic block which has loops or/and recursions will be decrypted multiple times. Therefore, by marking the number of loops and recursions in the table, we can prevent this problem. When a basic block has been called, the number of calls is reduced by one, and then if the number of calls became zero, the block should be re-encrypted from memory to protect against a memory dump. For example, in the case of loop, the source code is written in the C language, and its assembly code is shown in Fig. 11.

The second operand of the "*cmp*" command is 0Ah. It indicates the block "loc_401006" will be carried out 10 (=0x0A) times, and that is the number of loops or recursions.

In conclusion, we can summarize the algorithm of step 2 as

```
Procedure ConstructTable()
1.  entrypoint ← Find_EntryPoint(); // store an address
                                       of entry point
2.  currentPointer ← entrypoint;
3.  nextPointer ← currentPointer++;
4.  index ← 0;
5.
6.  while(File pointer is not end of file) {
7.     if(Current_opcode == jump or Current_opcode == branch){
8.    // branch or jump command is an unit of block
9.          nextAddress ← operand;
10.         // store an address of current address
11.         if(currentAddress == nextAddress) {
12.                 // loop, or recursion
13.            Tuple[index].Address ← currentAddress;
14.            Tuple[index].Size ← sizeofBlock;
15.            Tuple[index].Cnt ← prev_operand_2;
16.            Tuple[index].flag ← 0;
17.            //index, entry point address, size, number of calling,
                 and no protected key
18.            StoreAttribute(Tuple[index]);
19.         }
20.         else{
21.              if(FindAddress(Tuple[index].currentAddress) {
22.               // repeated calling
23.                  GenerateProtectedKey();
24.                  StoretoDatasection();
25.                  Tuple[index].flag ← 1
26.              }
27.         }
28.    }
29.    nextBlock ← Get_NextBlock(currentAddress);
30.    // Get next block's address
31.    currentAddress ← nextAddress;
32.    Sort(Tuple);
33.    Compute_blocksize(Tuple);
34.    CheckIterations();
35.    index++;
36. }
```

Fig. 12. Pseudo-code to construct indexed table.



Address (offset)	Block size	Number of calls	Flag
0x0040103E	19	1	0
0x00401051	9	2	1
0x0040105A	6	2	1
0x00401060	12	1	0
0x0040106C	8	2	1
	⋮		

Fig. 13. Example of constructing indexed table.

```
Procedure Decryption()
1. entrypoint ← Find_EntryPoint(); // store an address of
                                      entry point
2. currentAddress = entrypoint;       // initialization
3. nextAddress = 0;
4.
5. nextAddress = Find_next(entrypoint);   // find an address
                                      of next block in current block
6. Decrypt(IK, entrypoint);   // decrypt first block
7. Execute(currentAddress); // execute the first block
8.
9. while(File pointer is not end of file)
10. {
11.     if(LookupTableTable(nextAddress)) // if next block's flag
                                      in the indexed table indicates 1
12.     {
13.        random = ExtractRandomNumber(nextAddress, IK);
14.          // extract the random number with IK
15.        Decrypt(random, nextAddress); // decrypt with the
                                      random number
16.     }
17.     else                  // next block's flag indicates 0
18.       Decrypt(random, currentAddress);   // encrypt with the
                                      current block
19.
20. Execute(nextAddress); // execute the next block
21. ReEncrypt(currentAddress);
22.
23. }
```

Fig. 14. Pseudo-code to decrypt executable file.

the pseudo-code in Fig. 12. In addition, we can show an example of constructing the indexed table in Fig. 13.

The example code is composed of five basic blocks. The basic blocks are divided by "*jump*" or "*branch*" commands. At first, initialization is performed to construct the indexed table. 0x0040103E is set as the starting point of the program. After this, investigate the commands to find the "*jump*" or "*branch*." If the command is "*jump*" or "*branch*," store the operand of the command in the table because it becomes the first address of another block. In this example, 0x0040105A is stored in the table because of the command "*jne* 0x0040105A," which is at0x0040104F. The next address of the command becomes the first address of another block. So, 0x00401051 is stored in the indexed table. In this way, 0x0040106C and 0x00401060 are stored in order. At 0x0040106A, the command "*jmp* 0x00401051" is identified. 0x00401051 has been stored already, which indicates that there are multiple paths regarding the address 0x00401051. Hence, the block's information

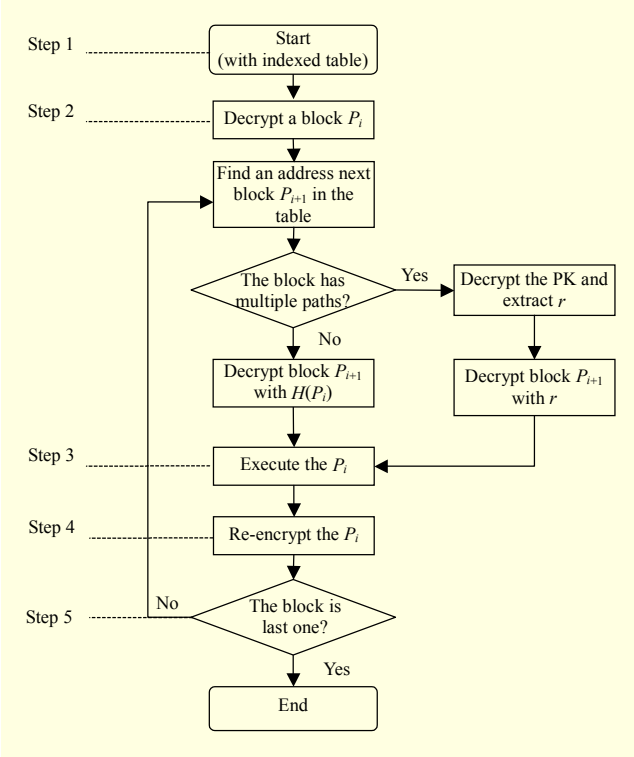


Fig. 15. Decryption process flow chart.

Table 2. Security comparisons between existing schemes and ours.

Security requirements	Cappaert's scheme	Jung's scheme	Proposed scheme
Confidentiality	O	O	O
Memory dumping prevention	O	O	O
Correct key chain	X	O	O
Tamper resistance	O	O	O

should be updated, and the random number should be generated as well. In this way, all the blocks can be identified.

3. Code Decryption

We can summarize the algorithm of all of the steps for the decryption process as the pseudo-code in Fig. 14.

The algorithm is divided into four steps: the lookup step, decryption step, execution step, and re-encryption step. These code decryption steps are described in the flow chart of Fig. 15.

At the beginning of the program's execution, the indexed table should refer to the entry point of the program, and then the encrypted block C_i should be decrypted into executable code P_i. Decrypted code P_i is executed after the block is decrypted using the preceding block's hash value. If a P_{i-1} has tampered into P'_{i-1}, the secret key will be $H(P_{i-1}) \neq H(P'_{i-1})$, so P_i cannot be decrypted properly. The indexed table contains the flag, which indicates whether the block uses random numbers or not. If the flag is 1, the encrypted block is decrypted by a random number. When the decrypted code completes the work, it is encrypted again and stored in the memory.

Table 3. Comparisons of increased size of executable file when block has n paths.

	Jung's scheme	Proposed scheme
Increased size of executable file	$(n-1) \cdot k$	l

IV. Analysis

This section begins with security analysis of the proposed scheme. Then, we show performance analysis and comparisons to clarify the advantage of our scheme. Lastly, we present the result of the experiment.

1. Security Analysis

To improve security, we adopted the indexed table based on a code encryption scheme. Cappaert's scheme does not meet the previously discussed security requirements. Our scheme meets the security requirements as described in Table 2.

The original binary code should be protected from static analysis and dynamic analysis by remaining confidential. Our scheme divides the program into basic blocks and then encrypts these basic blocks. It protects the confidentiality of the software. Our scheme achieves memory dumping prevention because our scheme decrypts only small parts of the code, so codes are not revealed all at once in the memory. The key chain is accomplished by using a random number. When a basic block invokes another block that is invoked by multiple blocks, a random number is used to decrypt the block. Cappaert's scheme did not satisfy the correct key chain requirement as previously discussed in section II. Tampering with a block is detected because the hash value of P_{i-1} was used as the secret key. If an attacker tampers with a plain or encrypted block, a corrupted code is generated at the later stage and thus the system will most likely crash due to an illegal instruction.

2. Performance Analysis

We analyze the efficiency when multiple paths occur. Cappaert's scheme did not consider multiple paths. Jung's

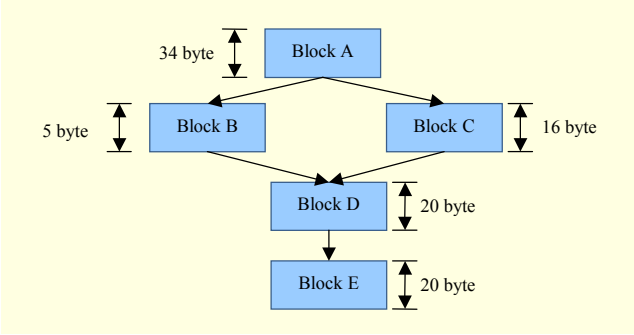


Fig. 16. Common target program.

Table 4. Performance comparisons between existing schemes and our scheme.

Feature	Cappaert's scheme	Jung's scheme	Proposed scheme
Original file size (B)	95	95	95
Number of blocks	5	5	5
Number of multiple paths	1	1	1
Decryption and re-encryption time (s)	-	0.0000857	0.0000446
C_t	-	4.28	2.23
C_s	-	1.02	0.84

scheme considered multiple paths, but it created additional overhead whenever it faced multiple paths because the invoked block was duplicated. We assume that there are n multiple paths. For example, in Fig. 10, D has 3 paths, thus n is 3. Additionally, we assume that k is a unit block's size, and l is the key size, which is originally a random number. At this time, when a block has n paths, the increased size of the executable file is as in Table 3.

The number of paths of a block n is always larger than 2. Thus, if k is larger than or equal to l, the increasing size of the executable file of Jung's scheme is always larger than in our scheme. Jung specified the k in his paper, and it was usually 16 B, 32 B, or 64 B. Consequently, if the size of a random number is specified as 16 B, the size of the executable file in Jung's scheme is always larger than in our scheme. If a program has additional multiple paths, the size differential between the schemes will increase.

We next compare the performance of our scheme with other schemes. Due to the fact that all of the schemes are implemented in different operating systems and test applications by authors of previous schemes, we assume a common target program as shown in Fig. 16. Although the structure of the program is very simple, it is used frequently in general software. The program has a total of 95 B and a 0.00002 s execution time. It is constructed with 5 blocks. In addition, it has one multiple path.

To compare, we use a method that is presented in [4]. Suppose that we have a program P and its modified version P'. Then, we define the time cost C_t and the space cost C_s as

$$C_t(P,P') = \frac{T(P')}{T(P)},$$

$$C_s(P,P') = \frac{S(P')}{S(P)},$$

where $T(X)$ is the execution time of program X, and $S(X)$ is its size. The result of the performance is shown in Table 4.

Cappaert's scheme cannot execute the common target program since the program has one multiple path. In Jung's scheme, the executable file size and execution time are increased since the scheme needs to convert a basic block to unit blocks and then duplicate the unit blocks. In this example program, 95 B codes are increased by 192 B, thus space cost C_s value is 1.02, and time cost C_t value is 4.28. In contrast with existing schemes, our scheme has 80 B of additional data, the space cost C_s value is 0.84, and the time cost C_t value is 2.23. C_s value and C_t value show that our scheme has better performance over Cappaert's scheme. In practice, the advantage of storage and time cost is useful in real time systems or in embedded systems which have constraints of performance.

3. Experiment

Next, we show the implementation and our evaluation of the proposed scheme. We implemented our scheme in the environment described in Table 5.

The process of the implemented program is shown in Fig. 17. Initially, an executable file is input into our program, and then the file is disassembled. The result of the process is an assembly code. Using this assembly code, our program constructs the indexed table and encrypts the codes. In the decryption process, the indexed table is used to decrypt the code.

Table 5. Implementation environment.

Feature	Description
Operating system	Windows XP Service Pack 3
Language	C/C++
Compiler	Microsoft Visual C++ 6.0
Cryptographic library	Win32 OpenSSL version 0.9.8
CPU	Intel Core2Duo CPU E7200
RAM	4 GB

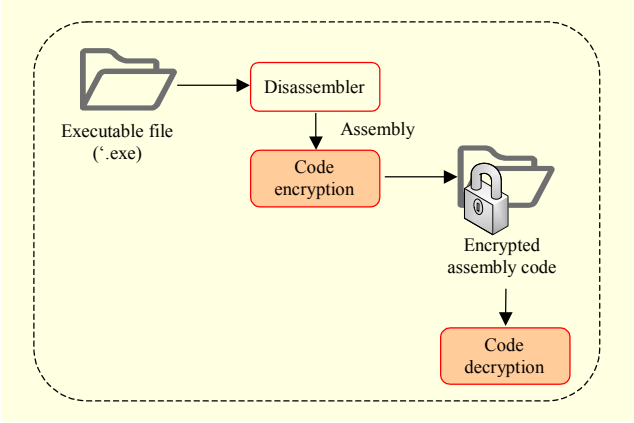


Fig. 17. Process of the implemented program.

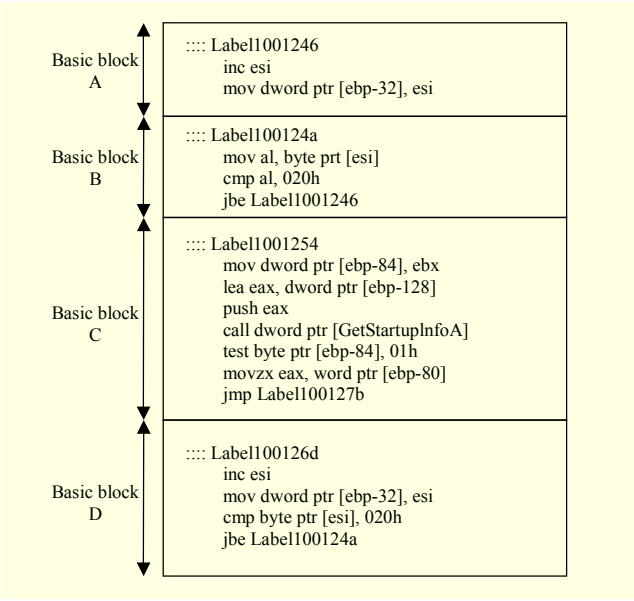


Fig. 18. Disassembled executable file.

We referred to PEDasm version 0.33, which is an open source disassembler. To measure the performance, we select three small programs which are default executable files in Windows XP, “systray.exe,” “regedt32.exe,” and “actmovie.exe”. “Systray.exe” is a background process which displays information, such as date and time. “Actmovie.exe” is used by some screensavers and Microsoft applications for video graphics, and “regedt32.exe” is a process associated with registry scanning. They are stored in “\$WINDOWS$\system32\”. We use a stream cipher, RC4 [17], as a cryptographic algorithm to encrypt and decrypt the code.

At first, the executable file is entered, disassembled, and divided into basic blocks as shown in Fig. 18. Then, the program performs table indexing and code encryption using the divided basic block. The results are shown in Fig. 19.

The result shows that the input program is “systray.exe”. The string “1414,” which is entered by the user, is hashed as SHA-1

```
Select the menu(1. Code Encryption  2. Code Decryption) :1

Enter the Initial Key : 1414

 == initKey ==
fb ce 66 f9 9c 80 92 83 63 8f 34 4e cb 3d 50 67 4e a6 41 89

Input file : systray.exe
Collecting directory entry symbols

=======Executable info=========
number of block : 18
8 blocks are called by multiple blocks

====================== Table ============================================
   Block index                Size     Number of calls         Flag
=========================================================================
     Fun1001098                278              1              0
     Fun10010c0                354              1              0
   Label10010f9                 98              1              0
   Label100110a                158              1              0
   Label1001126                 45              2              1
   Label100112f                 37              2              1
   Label1001135                 72              2              1
          main                 174              1              0
   Label1001159                103              2              1
   Label1001169                 65              1              0
   Label1001171                 36              2              1
   Label1001177                 78              1              0
   Label1001181                 35              2              1
   Label1001187                 19              2              1
   Label1001189                 59              3              1
   Label1001191                214              1              0
   Label10011ab                 23              1              0
   Label10011ae               1139              1              0
=========================================
Elapsed CPU time for encryption status:   0.483213949743  sec
```

Fig. 19. Result of program.

```
Select the menu(1. Code Encryption  2. Code Decryption) :2

Enter the Initial Key : 1414

 == initKey ==
fb ce 66 f9 9c 80 92 83 63 8f 34 4e cb 3d 50 67 4e a6 41 89

===Read from indexed table===
  278     1     0
  354     1     0
   98     1     0
  158     1     0
   45     2     1
   37     2     1
   72     2     1
  174     1     0
  103     2     1
   65     1     0
   36     2     1
   78     1     0
   35     2     1
   19     2     1
   59     3     1
  214     1     0
   23     1     0
 1139     1     0

==== Decrypt the code Completed======
Elapsed CPU time for normal status:   0.002410866553  sec
```

Fig. 20. Result of the lookup table and re-encryption.

[18] to generate the initial key *IK*. An executable file “systray.exe” has 18 blocks, and 8 blocks are invoked by multiple blocks. Hence, the 8 blocks have *PK*. Each block and related information is constructed as a table, and then the blocks are encrypted according to the table.

The constructed table is refereed to decrypt the code correctly, and then re-encryption is performed to protect the code from memory dumping. The results are shown in Fig. 20.

We measure and evaluate the encryption time and decryption time of three programs. At this time, we exclude external

Table 6. Results of measurement.

Feature	Regedt32.exe	Actmovie.exe	Systray.exe
Original file size (B)	3,584	4,096	3,072
Number of blocks	12	26	18
Number of multiple paths	7	11	8
Decryption and re-encryption time (s)	0.0016	0.0032	0.0024
C_t	6.680	6.119	10.985
C_s	1.027	1.674	1.047

libraries such as ".dll" files because they are implemented externally to the executable file. The result is shown in Table 6.

Table 6 shows programs increased sizes and execution times. *PK* and the constructed table make a program's size larger than before. Additionally, the execution time is increased due to table lookups and cryptographic operations. However, increases of data and execution time are less than in existing schemes.

V. Conclusion

This paper presented and discussed code encryption schemes for protecting software against reverse engineering and modification. Recently, Cappaert proposed a tamper-resistant code encryption scheme, and Jung proposed a key-chain-based code encryption scheme. However, Cappaert's scheme did not meet the security requirements for code encryption schemes, and Jung's scheme had an efficiency problem. Therefore, we proposed a new code encryption scheme based on an indexed table to guarantee secure key management and efficiency.

Finally, we implemented our scheme and measured the time cost and space cost. To improve efficiency, support from the compiler and operating system is needed [19].

References

[1] Business Software Alliance. Available: http://www.bsa.org

[2] B. Barak et al., "On the (Im)possibility of Obfuscating Programs," *Advances in Cryptology, LNCS*, vol. 2139, 2001, pp.1-18.

[3] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation–Tools for Software Protection," *IEEE Trans. Software Eng.*, vol. 28, no. 8, 2002, pp. 735-746.

[4] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–A Tool for Software Protection Research," *IEEE Security and Privacy*, vol. 1, no. 4, 2003, pp. 40-49.

[5] C.S. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Dept. of Computer Sciences, Univ. of Auckland, Tech. Report, no. 148, 1997.

[6] E. Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley Publishing, Inc., 2005.

[7] J.M. Memon et al., "A Study of Software Protection Techniques," *Innovations Adv. Techniques Computer Inf. Sci. Engin.*, 2007, pp. 249-253.

[8] J. Cappaert et al., "Toward Tamper Resistant Code Encryption: Practice and Experience," *LNCS*, vol. 4991, 2008, pp. 86-100.

[9] J. Cappaert et al., "Self-Encrypting Code to Protect Against Analysis and Tampering," *1st Benelux Workshop Inf. Syst. Security*, 2006.

[10] D.W Jung, H.S Kim, and J.G. Park, "A Code Block Cipher Method to Protect Application Programs From Reverse Engineering," *J. Korea Inst. Inf. Security Cryptology*, vol. 18, no. 2, 2008, pp. 85-96 (in Korean).

[11] G. Wroblewski, *General Method of Program, Code Obfuscation*, PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[12] D. Low, "Protecting Java Code via Code Obfuscation," *Crossroads*, vol. 4, no. 3, 1998, pp. 21-23.

[13] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *ACM Conf. Computer Commun. Security*, 2003, pp. 290-299.

[14] S. Chow et al., "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs," *LNCS*, vol. 2200, 2001, pp. 144-155.

[15] Y. Sakabe, M. Soshi, and A. Miyaji, "Java Obfuscation Approaches to Construct Tamper-Resistant Object-Oriented Programs," *IPSJ Dig. Courier*, vol. 1, 2005, pp. 349-361.

[16] Tao Zhang, Santosh Pande, and Antonio Valverde, "Tamper-Resistant Whole Program Partitioning," *Proc. Conf. Languages, Compilers, Tools Embedded Syst.*, vol. 38, 2003, pp. 209-219.

[17] R. Rivest, *The RC4 Encryption Algorithm*, RSA Data Security, Inc., Mar. 1992.

[18] NIST, "Secure Hash Standard," Fed. Inf. Process. Std., FIPS-180-1, Apr. 1995.

[19] M.R. Stytz and J.A. Whittaker, "Software Protection: Security's Last Stand," *IEEE Security Privacy*, vol. 1, no. 1, 2003, pp. 95-98.

Sungkyu Cho received his BS and MS in electrical and computer engineering from Sungkyunkwan University, Korea, in 2008 and 2010, respectively. He is currently an assistant engineer in Samsung Electronics. His research interests include cryptography, reverse engineering, and mobile security.

Donghwi Shin received his BS in physics from Sungkyunkwan University, Korea, in 2002, and his MS in computer science from Sungkyunkwan University, Korea, in 2008. He is currently enrolled in a PhD course for information security in Sungkyunkwan University. He is a senior researcher at Korea Internet and Security Agency. His research interests include malware analysis, reverse engineering, and exploit development.

Heasuk Jo received her BS in computer engineering from Hansung University, Korea, in 2003, and MS in computer engineering from Sungkyunkwan University, Korea, in 2005. She is currently enrolled in a PhD course for electrical and computer engineering in Sungkyunkwan University. Her research interests include cryptography, information security and assurance, and mobile security.

Donghyun Choi received his BS and MS in electrical and computer engineering from Sungkyunkwan University, Korea, in 2005 and 2007, respectively. He is currently enrolled in a PhD course of mobile systems engineering in Sungkyunkwan University. His research interests include cryptography, SCADA, mobile security, and DRM.

Dongho Won received his BE, ME, and PhD from Sungkyunkwan University in 1976, 1978, and 1988, respectively. After working at ETRI from 1978 to 1980, he joined Sungkyunkwan University in 1982, where he is currently a professor in the School of Information and Communication Engineering. His interests are in cryptology and information security. In 2002, he was the president of the Korea Institute of Information Security and Cryptology.

Seungjoo Kim received the BS, MS, and PhD in information engineering from Sungkyunkwan University, Korea, in 1994, 1996, and 1999, respectively. Prior to joining the faculty at Sungkyunkwan University in 2004, he was the director of the cryptographic technology team and the IT security evaluation team of Korea Internet and Security Agency for five years. Currently, he is an associate professor of the School of Information and Communication Engineering at Sungkyunkwan University. His research interests include cryptography, information security, information assurance, and digital forensics.