

# A Goal-Oriented Approach to Software Obfuscation

Hiroki Yamauchi,<sup>†</sup> Akito Monden,<sup>†</sup> Masahide Nakamura,<sup>††</sup>  
Haruaki Tamada,<sup>†††</sup> Yuichiro Kanzaki<sup>††††</sup> and Ken-ichi Matsumoto<sup>†</sup>

<sup>†</sup>Nara Institute of Science and Technology, 8916-5 Takayama Ikoma Nara, Japan

<sup>††</sup>Kobe University, 1-1 Rokkodai-cho, Nada-ku, Kobe, Japan

<sup>†††</sup>Kyoto Sangyo University, Motoyama Kamikamo Kyoto-Shi Kyoto, Japan

<sup>††††</sup>Kumamoto National College of Technology, 2659-2 Suya Koshi Kumamoto, Japan

## Summary

Various software obfuscation techniques have been proposed. However, there are few discussions on proper use of these obfuscations against imaginable threats. An ad-hoc use of obfuscations cannot guarantee that a program is sufficiently protected. For a systematic use of obfuscations and the verification of the result, this paper proposes a goal oriented approach to obfuscation. Specifically, we (1) define the capability of an imaginary cracker, (2) identify the cracker's goal, (3) conduct a goal-oriented analysis, (4) select obfuscations to disrupt all sub-goals, and (5) apply selected obfuscations to the program. As a case study, we define a security goal and a threat model for a Java implementation of a cryptomeria cipher (C2) program, and then, based on the model, we demonstrate how the goal oriented analysis is conducted and obfuscation techniques are applied to places where they are needed.

### Key words:

*Software Protection, Reverse engineering, Secret Hiding, Program Analysis*

## 1. Introduction

A variety of confidential information is present in a typical software product, such as 1) subroutines and algorithms that are valuable trade secrets and/or related to system security, *e.g.* decryption algorithms of digital rights management (DRM) systems [1]; 2) constant values and text strings related to system security, *e.g.* decryption keys of DRM systems [4][5]; 3) internal function points, *e.g.* branching statements for license checking [17]; 4) internal interfaces, *e.g.* function call statements to a secure library module; and 5) external interfaces, *e.g.* secret "service entrances" that provide full access to the systems [13]. Many computer systems have been "cracked", with serious damages to the producers and/or suppliers of the systems. For example, secret codes for the CSS encryption standard for DVD media were revealed in 1999 [28].

In order to hide secrets in software implementation, a number of obfuscation techniques have been proposed including lexical obfuscations (*e.g.*, comment removal, identifier renaming and debugging info removal, etc.) [33], data obfuscations [7][9][29], and control-flow obfuscations [6][25][35]. These obfuscation techniques transform a program so that it becomes more difficult to understand, yet is functionally equivalent to the original one [8].

However, there is no systematic method on how to apply obfuscation techniques appropriately. So, it is unclear which obfuscation technique should be used. Also, it is unclear which portion of the program should be obfuscated, and how much effects of obfuscation can be expected. These problems are caused because many obfuscation techniques do not count the purpose and the target of a cracker enough.

The research objective of this paper is to establish a goal-oriented analysis framework for proper use of existing obfuscation techniques. Our key idea is to assume an imaginary cracker with his/her explicit purpose (goal) and the target (program), then break the goal down into pieces, each of which an appropriate obfuscation is applied to.

The remainder of the paper first describes the status quo of software obfuscation (Section 2). Next, proposes a framework for goal-oriented analysis to software obfuscation (Section 3). Afterwards, we describe a case study to apply the proposed framework to a practical cipher program (Section 4); and in the end, summary will be shown (Section 5).

## 2. Software Obfuscation Techniques and Their Problems

From a perspective of selecting proper obfuscation techniques, we need to consider the type of target to which each obfuscation technique can be applied. Many obfuscation techniques are generic ones, *i.e.* they are

applicable to any types of programs. On the other hand, there exist strong but target-specific obfuscations. Below introduces each type and its problems.

### 2.1 Generic Obfuscation Techniques

Most obfuscation techniques are generic ones, e.g. control flow obfuscation [6][25][35], inter-module call relation obfuscation [26], replacing a high-level instruction with a set of low-level instructions [20], inserting opaque predicates [10], transforming data structures [9], renaming identifiers [33], data obfuscation using homomorphism [3][27], use of self-modifying code [16], etc. These obfuscation techniques are focusing on building a “complex” program rather than preventing a cracker’s actions to achieve his/her goals. Many of these are light weight, i.e. applicable to resource-limited environment because of their small performance penalty.

Unfortunately, it is unclear how effective these obfuscations are in protecting confidential information inside a program. It is because “complex” is not equal to “difficult to crack”. In addition, many of obfuscations do not assume a clear threat model, a model of cracker’s behavior to break the security.

### 2.2 Target Specific Obfuscation Techniques

One of the most powerful obfuscation methods for protecting cipher software is a *white-box cryptography*, which was originally proposed by Chow et al. [4][5] and now became a hot research topic [12][18]. In this method, a set of computations using a secret key are replaced with lookup tables so that the key is hidden inside the tables and becomes unrecognizable to the cracker.

However, although this method can add strong protection to a specific cipher program, its application area is quite limited since it requires a large memory space (several megabytes) and imposes a serious performance penalty. Therefore, especially for resource-limited environments, such as mobile devices, we need an alternative way to protect cipher programs from crackers.

### 2.3 Toward Systematic Application of Obfuscation

Since the application area of target specific techniques are limited, we focus on the light weight, generic obfuscation techniques to protect any program containing security-sensitive data. To identify which obfuscation technique to be employed and which portion of the program to be obfuscated, we need to clarify an imaginary cracker’s capability, and also, his/her potential activities of program analysis, so that we are ready to select a set of obfuscation techniques to disrupt each potential activity.

## 3. A Framework for Goal-Oriented Analysis

### 3.1 Basic Idea

Figure 1 shows a concept of the proposed approach in contrast to the conventional approach to software obfuscation. The conventional approach in Figure 1 (a) takes a program  $P$  as an input and conducts a generic obfuscation  $O(x)$  to produce an obfuscated program  $P'$ . In this approach,  $O(x)$  should be responsible to protect secrets included in any given  $P$  against any imaginary crackers; however, such responsibility is not enough considered in conventional generic obfuscation techniques. In contrast, the proposed approach in Figure 1 (b) takes both  $P$  and a cracker model as inputs and conduct a goal oriented analysis to produce a *goal tree*, which connects a cracker’s goal with its sub-goals. Then, the proposed approach applies existing obfuscation techniques  $O_1(x) \dots O_n(x)$  to proper places in  $P$  to disrupt all sub-goals in the goal tree.

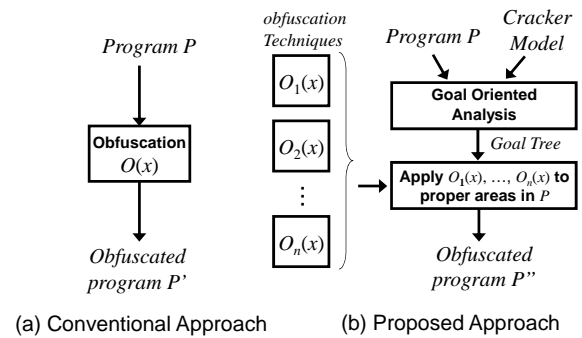


Fig. 1 A concept of the proposed method

In our previous work, we proposed a cracker-centric approach to give a guideline for employing existing obfuscation method to disrupt cracker’s actions [37]. However, the paper did not discuss the systematic application of obfuscation techniques. Consequently, in this paper we propose a goal-oriented approach to systematically protect confidential information in a program. The key idea is to assume an imaginary cracker with his/her purpose and target and to break down the goal into pieces, each of which an appropriate obfuscation is applied to. To implement the key idea, at first, we determine a capability of an imaginary cracker. Next, we identify a cracker’s goal, and conduct a goal oriented analysis. The goal oriented analysis is to decompose and to refine the goal into more specific sub-goals and repeat the decomposition until no sub-goal can be further decomposed. Finally, we select an obfuscation technique to prevent understanding for every terminal sub-goal.

Our goal-oriented approach consists of the following five steps.

**Step1.** Define the capability of an imaginary cracker.

**Step2.** Identify a cracker's goal.

**Step3.** Conduct a goal-oriented analysis.

**Step4.** For every terminal sub-goal, select obfuscation.

**Step5.** Apply the selected obfuscations to the program.

From next subsection, we describe details of each step.

### 3.2 Define the Capability of an Imaginary Cracker

When we consider employing a security mechanism to achieve any security goal, we must define a realistic threat model, a model of what a cracker is able (and not able) to do in the real world. For example, a cracker may have an executable (binary) program and an understanding of the principles of an algorithm used in the program. Also, it will be reasonable to assume that the cracker has a static analyzer such as a disassembler and a decompiler, as well as a dynamic analyzer (debugger) with "breakpoint" functionality.

In [24], Monden et al. characterized a cracker's knowledge and resources along three dimensions, (1) understanding level of a protection mechanism being used, (2) skill level of system observation, and (3) skill level of system control. These dimensions seem useful for evaluating any software protection mechanism; however, in this paper, the dimension (1) is presently out of the scope since we do not assume any particular protection method yet. Besides the dimension (1), we need to characterize the cracker's knowledge (understanding level) of a target system.

Based on the discussion above, this paper characterizes the capability model of an imaginary cracker, from three dimensions: (A) knowledge, (B) observation and (C) control. Below describes a skeleton of capability description for each dimension assuming a very skillful cracker.

#### (A) Knowledge

The cracker has full knowledge of the principles and external specification of a program.

#### (B) System Observation

The cracker owns a binary file, disassembled code and/or decompiled code of a target program  $P$ , as well as a computer system  $M$  in which  $P$  is executed. The cracker has a debugger with breakpoint functionality that can observe internal states of  $M$ , e.g. memory snapshot of  $M$ , audio-visual outputs of  $M$  and the input and output value of  $P$ . The cracker also observes the execution trace of  $P$ , i.e. a history of executed opcodes, operands and their values.

#### (C) System Control

The cracker operates the keyboard and mouse inputs of  $M$ , as it executes  $P$  with an arbitrary input. The cracker can change the instructions in  $P$  as well as the operand values and the memory image of  $M$  in any desired way, before and/or during running it on  $M$ .

Under the capability description above, crackers have various avenues of attack. They might inspect disassembled code of  $P$  and find a portion of code that implements a particular part of an algorithm being used in  $P$ . They also might observe a stack memory to find candidates of a secret data pushed onto the stack memory during execution [2]. Furthermore, they might collect multiple execution traces of different inputs, and compares them to find candidates of a fixed data appeared in operand values that are insensitive to the inputs [36].

### 3.3 Identify Cracker's Goal

In the second step of our analysis, we identify cracker's goal. For the given program  $P$ , we define a specific goal, for which the cracker reverse-engineer  $P$ .

Here we assume of protecting a typical cipher program in which DES algorithm was used (Figure 2). The input of the program is a cipher text sequence, which is an encrypted digital media content. The output is a clear text sequence, which is the decrypted audio or video media.

As shown in Figure 2, DES is a Feistel network-based block symmetric cipher. Inside the program, the cipher text is split into left (upper bits) and right (lower bits). Then, from the secret key  $K$ , the key scheduler generates a round key  $k_n$  for each decryption round. Afterwards, the Feistel function de-scrambles the cipher text using the round key. This step is repeated 16 rounds, and finally the data is decrypted.

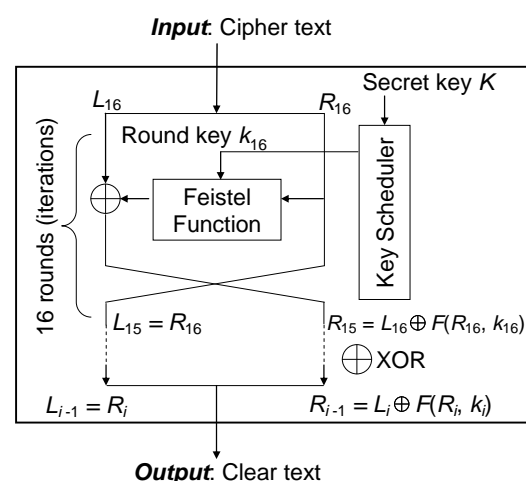


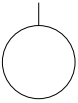
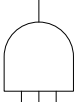
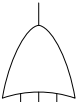




Fig. 2 Overview of DES algorithm

There are several critical data that must be protected from crackers in DES program: (1) the secret key  $K$ , (2) round keys  $k_1, \dots, k_{16}$  and (3) secret data table often included in Feistel function. The cracker's goal would be to extract these secret information from the program. For simplicity, this paper assume extracting (2) round keys as the goal to explain our idea in the following subsections.

Table1: Goal tree symbols

	<b>Root goal</b> The final goal of a cracker in attacking the target system.
	<b>Intermediate goal</b> A sub-goal decomposed from the parent node (root goal or an intermediate goal.) A cracker needs to complete intermediate goals before achieving the root goal.
	<b>Terminal goal</b> A primitive sub-goal that cannot be decomposed further.
	<b>AND gate</b> A gate that indicates all lower goals must be completed to achieve the higher goal.
	<b>OR gate</b> A gate that indicates either one or more goals must be completed to achieve the higher goal.
	<b>Transfer in</b> A transfer node connected to a "transfer out" node of other goal tree (child tree).
	<b>Transfer out</b> A transfer node connected to a "transfer in" node of other goal tree (parent tree).

### 3.4 Conduct a Goal-Oriented Analysis

Given a cracker's goal, such as "finding round keys", a cracker may have several avenues (sub-goals) to achieve the goal. Also, there would be smaller sub-sub-goals that need to achieve each sub-goal. In our analysis, we describe such goal breakdown structure as a goal tree (an AND-OR graph).

Table 1 describes symbols we use to build a goal tree. These symbols were typically used in Fault Tree Analysis (FTA) [34] to decompose a failure into its possible causes. As shown in Table 1, we have three types of goals: (1)

root goal, (2) intermediate goals and (3) terminal goals. These goals are connected one another by either an AND gate or an OR gate. If a goal is connected to lower (intermediate or terminal) goals by an AND gate, this means the cracker needs to complete all the lower goals to achieve higher goal. On the other hand, if a goal is connected to lower goals by an OR gate, the cracker needs to complete either one of lower goals. In addition, if the goal tree grew too large to draw in a limited space, symbols "transfer in" and "transfer out" could be used to divide the tree into parts.

Below describes a basic (top-down) procedure to build a goal tree:

- (1) Set the root goal at the top of a goal tree. Figure 3 describes an example of a goal tree having "find round keys" as a root goal.
- (2) Break the goal down to intermediate goals that a cracker might find based on Cracker's Capability Model defined in Section 3.2. In Figure 3, the goal "find round keys" is decomposed into two intermediate goals "Find keys in F-function" and "Find keys in key scheduler" because round keys appears in both F-function and the key scheduler.
- (3) Connect the relation between identified goals and the higher goal either by an AND gate or an OR gate. In case of Figure 3, intermediate goals "Find keys in F-function" and "Find keys in key scheduler" are connected to the higher goal by an OR gate.
- (4) For each identified (intermediate) goal, if there seems no sub-goal that contributes to achieve the goal, then set the goal as a terminal goal in the tree. Otherwise, Repeat (2) ... (4) until all the leaf goals become terminal ones.

The procedure described above is a top down approach, which is a straight forward way to build a goal tree; however, using the bottom up approach together with the top down approach would be helpful to build the tree. Below describes the bottom up approach to supplement the top down approach.

- (1) Identify any clues that the cracker might find in the target program based on Cracker's Capability Model. Here, a clue means a piece of information that might contribute to achieve the root goal. For example, an operator "XOR" can be a clue since a lot of XORs are expected to appear in a DES cipher routine.
- (2) Identify abstract clues or intermediate goals that the cracker might find based on primitive clues identified in (1). For example, "F-function" can be an abstract clue because F-function contains a lot of XOR operators. Since "Locate F-function" is an intermediate goal identified in the top down approach (Figure 3), "Finding operator XOR" is connected to it as a lower goal.

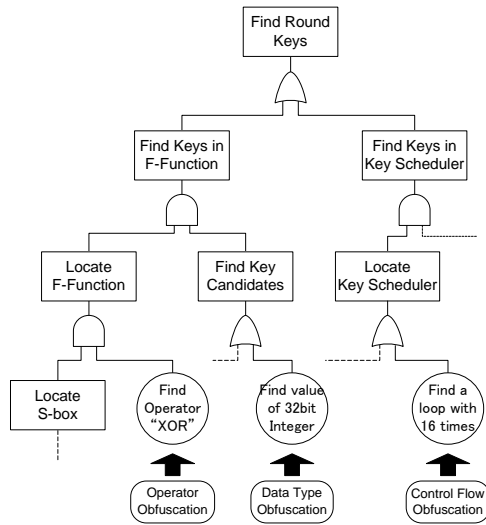


Fig. 4 Select an appropriate obfuscation

### 3.5 Select an Appropriate Obfuscation

For each terminal goal identified in Section 3.4, we select an appropriate obfuscation to disrupt the cracker achieving the terminal goal. Figure 4 shows an example of selecting obfuscation techniques. In this example, in order to prevent the cracker from “finding the operator XOR”, we employ “operator translation obfuscation”. Also, to disrupt finding a 32 bit integer value, we employ data type obfuscation (e.g. splitting 32 bit integer variable into four 8 bit variables). Similarly we employ a control flow obfuscation to hide “16 times loop”.

### 3.6 Apply Selected Obfuscations

We apply all the obfuscations selected in Section 3.5 to the target program. As a result all terminal goals become difficult to achieve, which means all inter-mediate goals become difficult to achieve. Therefore, the root goal “find round keys” also becomes difficult to achieve.

## 4. Case Study

To explain how to apply our approach to an actual program and to show how it works, this Section introduces a case study to obfuscate a cipher program typically used in DRM systems.

### 4.1 Target Program

Here we assume of protecting a typical DRM program in which *cryptomeria cipher* (C2) algorithm was used. Overview of C2 algorithm is described in Figure 5, and

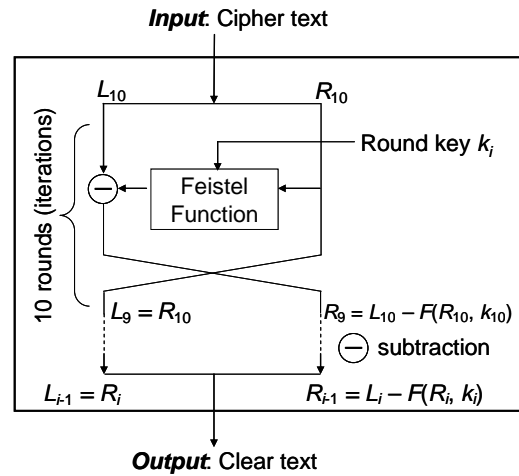


Fig. 5 Overview of C2 algorithm

the source program (written in Java) to be obfuscated is shown in Figure 8. This algorithm is used in CPPM (Content Protection for Pre-recorded Media) / CPRM (Content Protection for Recordable Media) scheme [1].

As shown in Figure 5, C2 is a Feistel network-based block symmetric cipher just like DES. The box “F” indicates the Feistel function. Figure 6 shows details of the Feistel function. The major distinction point is that, C2 uses arithmetic addition and subtraction while DES does not. (This paper assumes that the C2 program does not contain a key scheduler function because we assume using ECB cipher mode (which does not require the key scheduler) and also we wanted to keep the example simple.

### 4.2 Crackers’ Goal and Capability Model

#### 4.2.1 Knowledge

Since the specification of C2 algorithm is open to public [1], the cracker’s goal here is to find all round keys in Figure 8 so that the cracker is able to write his/her own program that can decrypt existing DRM media contents.

By reading the C2 specification [1], the cracker will understand the C2 algorithm (Figure 5) and obtains the following knowledge.

- Round keys  $k_i$  are either supplied from a key schedule routine or directly written in  $P$  as constant values. In the former case, there exists a device key  $K$  in  $P$ , and  $K$  is supplied to the key schedule routine. In the latter case, both  $K$  and the key schedule routine may not exist in  $P$ . In this case, the cracker’s goal is to find all the round keys (this is the case of this paper.)
- The number of rounds (iterations) is 10. So, there are 10 round keys  $k_1 \dots k_{10}$ .

- The length of each round key is 32 bit.
- The input block size is 64 bit. A block is divided into  $L$  (upper 32 bit) and  $R$  (lower 32 bit).
- There is a Feistel function  $F$  in  $P$ . Either  $L$  or  $R$  is input to  $F$  in each round.
- There is a subtraction expression right after  $F$ .

Similarly, as for the Feistel function  $F$  (Figure 6), the cracker should have the following knowledge.

- Either  $L$  or  $R$  is added to a round key. This indicates that there exists in  $P$  an “add” opcode that takes two 32 bit operands.
- The result of addition  $X$  (32 bit) is divided into four 8 bit blocks  $x_1 \dots x_4$ . It can be guessed that this division is done by statements “ $x_1 = (X \ggg 24) \& 0xff$ ,  $x_2 = (X \ggg 16) \& 0xff$ ,  $x_3 = (X \ggg 8) \& 0xff$ ,  $x_4 = X \& 0xff$ .”
- The lowest 8 bit block  $x_4$  is translated by S-box table. Here, the S-box table is a set of 256 8 bit values. This indicates there is an array having 256 elements in  $P$ . The translation can be done by a reference to the array, e.g. “S-box\_array[ $x_4$ ].”
- Remaining 3 blocks  $x_1 \dots x_3$  are XOR’ed with 0xc9, 0x2b, and 0x65, respectively. This indicates there exist XOR expression and constant values 0xc9, 0x2b and 0x65 in  $P$ . Afterwards, these 3 blocks are rotated leftward 2 bit, 5 bit, and 1 bit, respectively.
- Then, four blocks are concatenated back to 32 bit value. This value is then XOR’ed with 9 bit rotated value and 22 bit rotated value. This indicates there exist “shift” opcode and constant values 9 and 22 in  $P$ . It can be guessed that the concatenation was done by an expression “ $x_1 \ll 24 \mid x_2 \ll 16 \mid x_3 \ll 8 \mid x_4$ .”

#### 4.2.2 Observation and Control

The cracker uses disassemblers and decompilers to statically analyze the program. Sun Microsystems provides java disassembler (known as “javap -c” command). Also, Java disassembler D-Java, which produces jasmin format assembly code [21] is provided by Meyer [22]. Disassembled code can be re-assembled back to class files by jasmine assemblers [23][30]. Various java decompilers are also available although in most cases, they produce imperfect java source code [32]. The cracker also uses debuggers, such as jdb, provided by Sun Microsystems, IDA Pro [14] by Hex-Rays, Omniscient Debugger [15] by Bil Lewis.

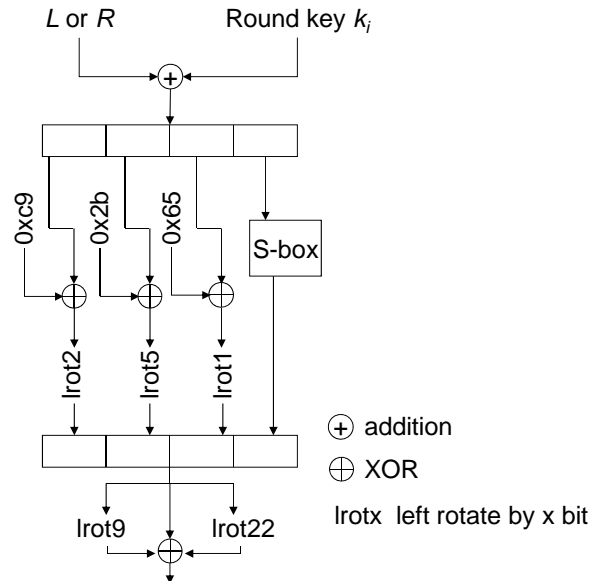


Fig. 6 Feistel function of C2

#### 4.3 Goal-Oriented Analysis

We assume that the cracker’s root goal is to “find round keys” in the C2 program (Figure 8) that needs to be obfuscated. Based on this root goal, we conducted a goal-oriented analysis and built a goal tree (Figure 7). Below describes an example of a top-down approach to build a goal tree.

There are two potential avenues to achieve the root goal “find round keys”: one is to “find key candidates in constants” via static analyses and the other is to “find key candidates in variables” via dynamic analyses (in addition to static analysis). These two avenues were set as sub-goals of the root goal in Figure 7.

To achieve the sub-goal “find key candidates in constants”, we focus on the characteristics of constants where the key candidates could be found. Such characteristics can be considered as “clues” that might contribute to achieve the goal. As shown in Section 4.2.1, each key is 32 bit length. Also in the actual program (Figure 8), key values are assigned to an array  $sk[]$  as a set of 32 bit constants by “int  $sk[] = \{ 0x789ac6ee, 0x79bc3398, \dots \}$ ”. These 32 bit constants could be identified by disassembling the program, although not all 32 bit constants in the disassembled code are round keys. Therefore, “find 32bit integers” can be a lower goal of “find key candidates in constants”. We consider this a terminal goal, which cannot be decomposed further.

Now we decompose the other avenue “find key candidates in variables”. There are two areas in the

program where a variable potentially possess a key value: (1) inputs of F-Function and (2) key manipulation if F-Function. In the actual program (Figure 8), a key value hold the variable *sk[]* is passed to the variable *key* in the F-Function. Also, inside the F-Function, the variable *key* is used to proceed with the decryption. These avenues could be discovered by the cracker based on the knowledge described in Section 4.2.1 and the capability in Section 4.2.2. Therefore, we set two sub-goals “inspect inputs of F-Function” and “inspect key manipulation in F-Function” as lower goals of “find key candidates in variables”.

For both sub-goals “inspect inputs of F-Function” and “inspect key manipulation in F-Function”, the cracker needs to find the F-Function. Therefore, we set lower goal “find the Feistel function” in Figure 7.

We consider two types of clues that might contribute to achieve the goal “find the Feistel function”: (1) clues related to function calls for F-Function and (2) clues that comes from the characteristics of F-Function itself. As for the former clues, “10 times loop” and “SUB operation” are big clues since F-Function is called 10 times in the C2 algorithm and SUB operation is required right after calling F-Function as shown in Figure 5. As for the latter clues, we focus on the distinctive items in the F-Function: (1) distinctive values, (2) ADD operator, (3) S-box, (4) 32 bit rotate function, (5) 8 bit rotate function and (6) concatenate function. When the cracker finds one of these clues in a certain program area, he/she may think there might be the F-Function in that area. And, the cracker may try to find other clues in the same (or neighborhood) area so that he/she becomes more confident that he/she surely found the F-Function. Thus, all these clues should be included in the goal tree so that they are obfuscated in the later step of our framework.

Further goal decompositions are shown in Figure 7. For each remaining sub-goal, we inspected the possibility of further goal decomposition, by thinking of any smaller actions to achieve the sub-goal, or by thinking of any clues that might contribute to achieve the sub-goal. If no more decomposition was available, then we set the goal as a terminal goal. Finally, 27 sub-goals, including 12 intermediate sub-goals and 15 terminal sub-goals, have been identified as shown in Figure 7.

#### 4.4 Select an Appropriate Obfuscation

Due to limited space, here we focus on four sub-goals in the goal tree (labeled “A”, “B”, “C” and “D” in Figure 7) to be obfuscated.

##### 4.4.1 Obfuscating Distinctive Values

Here we try to disrupt the cracker’s action to achieve the sub-goal “identify distinctive values 0x65, 0x2b, 0xc9” (labeled “A” in Figure 7.) This sub-goal contributes to

achieve the upper goal “find Feistel function”, because constant values 0x65, 0x2b and 0xc9 are expected to appear in the Feistel function of C2 algorithm (Figure 6).

In Figure 8, a program to be obfuscated, these constant values appears in the function “public static int F(int data, int key)”. In this function 0x65 appears in the statement:

```
u = (byte)(v[0] ^ 0x65);
```

To hide 0x65, we split it into two values 0x21 and 0x44, which satisfy “0x65 = 0x21 XOR 0x44.” By using these two values we could replace the statement with the following two statements:

```
u = (byte)(v[0] ^ 0x21);
u = (byte)(u ^ 0x44);
```

Similarly, we hid other two constants 0x2b and 0xc9 by exploiting the relations “0x2b = 0x28 XOR 0x03” and “0xc9 = 0x41 XOR 0x88.” The resulting obfuscation is labeled “A” in Figure 9.

##### 4.4.2 Obfuscating 10 Times Loop

Next we try to disrupt achieving the sub-goal “identify 10 times loop” (labeled “B” in Figure 7.) This sub-goal also contributes to achieve the upper goal “find Feistel function”, because F-Function is called 10 times in the C2 algorithm (Figure 5). In our example, we simply applied loop unrolling to remove the 10 times loop (label “B” in Figure 9).

##### 4.4.3 Obfuscating Concatenate Function

Next we obfuscate the concatenate function (label “C” in the goal tree of Figure 7). A concatenate function is to concatenate four 8 bits values into a 32 bits value. The simplest implementation is  $t = v[3] \ll 24 \mid v[2] \ll 16 \mid v[1] \ll 8 \mid v[0]$  where *v* is an array contains four 8bit values, and *t* is the resultant value. To identify the concatenate function, a cracker would find clues such as values 24, 16 and 8, bit shift operator “<<” and OR operator “|”. Actually, in Figure 8, the target of obfuscation is implemented as the following statements:

```
t = (int)v[3] << 24 | (int)v[2] << 16 | (int)v[1] << 8 |
(int)v[0];
```

To remove bit shift operators “<<”, we used multiplication instead. The leftward *n* bit shift can be replaced by multiplication by  $2^n$ . For example,  $v[3] \ll 24$  can be replaced by  $v[3] * 16777216$ . By this replacement, the distinctive value “24” is also removed (replaced by  $16777216 = 2^{24}$ )

To remove OR operators “|”, we exploited the De Morgan’s law “ $P \mid Q = \sim(\sim P \ \& \ \sim Q)$ ” so that OR is

replaced by NOT and AND operators. As a result we obtained:

$$t = \sim(((\sim((\text{int})v[3] * 16777216) \& \sim((\text{int})v[2] * 65536) \& \sim((\text{int})v[1] * 256)) \& \sim((\text{int})v[0]));$$

#### 4.4.4 Obfuscating 32 Bit Integers

Next we obfuscate 32 bit integers, which implement round keys (label "D" in Figure 7). The cracker would try the brute force attack and monitor every 32 bit value appearing in the assembly code or in the stack [2] to find the round key. To protect the key from the attack, the key value must be transformed.

In Figure 9 (label "D"), a homomorphic transformation by a linear function [29] was applied to key values  $sk[]$ . This method transforms each data by a linear function (we used  $f(x) = 4x + 3$ .) By this transformation, a key 0x789ac6ee was encoded as 0x3ffb81617L (=  $4 * 0x789ac6ee + 3$ ). This encoded key is then used in the Feistel function in the statement:

$$t = \text{data} + \text{key};$$

To compute with the encoded key, the statement "t = data + key" is replaced with "t = data \* 4 + key" so that resultant value t holds an encoded value. In the later, when clear text of t is required, we compute inverse transformation  $f^{-1}(x) = (x - 3) / 4$  so that t becomes non encoded value. The resultant program is shown in Figure 9.

Important things here is that, during the above computation, the original key value 0x789ac6ee appears neither in constants, variables nor the stack. Instead of using linear encoding, we could also consider using residue encoding, bit exploded encoding [7], secret sharing homomorphism [3][27], variable merging [11] and the use of error collecting code [19] to hide the key value.

After transforming keys by a homomorphic transformation or by other techniques, transformed keys should be implemented as a set of smaller (e.g. 8bit) sub keys so that it becomes more difficult to find key candidates for the cracker.

#### 4.5 Hiding Obfuscation

After all obfuscations were applied, i.e. all the paths from leafs to the root of the goal tree were blocked by obfuscations, we need to carefully inspect the obfuscated program if there exists any clue to discover the secret.

Since obfuscation methods themselves have distinctive features, we need to hide them so that the cracker can not recognize which obfuscation method is being used. For example, if we employ residue encoding [7] to hide round keys, a lot of modulo opcodes will be introduced in the obfuscated program. In this case, we need to write our own (obfuscated) modulo routines instead of simply using module opcodes.

## 5. Summary

This paper proposed a goal-oriented framework of applying existing obfuscation techniques to hide any secret in a program. This paper also demonstrated how the proposed framework could be applied to a cipher program through a case study with C2 cipher.

Our future work is to evaluate the proposed framework with other programs and to develop a guideline for building a goal tree and for selecting appropriate obfuscations.

## Acknowledgments

The work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Grant-in-Aid for Scientific Research (C), 19500056 and Grant-in-Aid for Young Scientists (B), 18700062.

## References

- [1] 4C Entity, "CPRM Common Cryptographic Functions" in Content protection for recordable media specification – Introduction and common cryptographic elements rev. 1.01, pp. 15-18, May. 2007.
- [2] K. Akai, M. Misawa, and T. Matsumoto, "Evaluating tamper resistance by searching runtime data," *IPSI Journal*, Vol.43, No.8, pp.2447-2457, Aug. 2002. (in Japanese).
- [3] J. C. Benaloh, "Secret sharing homomorphisms: keeping shares of a secret," *Proc. Advanced in Cryptology*, pp. 251-260, 1987.
- [4] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot, "A white-box DES implementation for DRM applications," *Proc. 2nd ACM Workshop on Digital Rights Management (DRM2002), Lecture Notes in Computer Science*, Vol. 2696, pp. 1-15, 2003.
- [5] S. Chow, P. Eisen, H. Johnson and P. van Oorschot, "White-box cryptography and an AES implementation," *Proc. 9th International Workshop on Selected Areas in Cryptography (SAC2002), Lecture Notes in Computer Science*, Vol. 2595, pp. 250-270, 2003.
- [6] S. Chow, H. Johnson, and Y. Gu, "Tamper resistant control-flow encoding," *United States Patent 6,779,114*, Filed 19 Aug. 1999, Issued 17 Aug. 2004.
- [7] S. Chow, H. Johnson, and Y. Gu, "Tamper resistant software encoding," *United States Patent 6,594,761*, Filed 9 June 1999, Issued 15 Jul. 2003.
- [8] C. Collberg, and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation – Tools for software protection," *IEEE Trans. on Software Engineering*, Vol. 28, No. 8, pp. 735-746, 2002.
- [9] C. Collberg, and C. Thomborson, D. Low, "Breaking abstractions and unstructuring data structures", *Proc. IEEE International Conference on Computer Languages (ICCL98)*, pp. 28-38, May 1998.
- [10] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," *Proc.*



- ACM Symposium on Principles of Programming Languages (POPL98)*, Jan. 1998.
- [11] C. Collberg, C. Thomborson, and D. Low, "Obfuscation techniques for enhancing software security," *United States Patent* 6,668,325, Assignee: InterTrust Inc., Filed 9 June 1998, Issued 23 Dec. 2003.
- [12] M. Jacob, D. Boneh, and E. Felten, "Attacking an obfuscated cipher by injecting faults," *ACM Workshop on Digital Rights Management, Lecture Notes in Computer Science*, Vol. 2696, pp. 16-31, 2003.
- [13] J. Havrilla, "Borland/Inprise Interbase SQL database server contains backdoor superuser account with known password," US-CERT, Vulnerability Note VU#247371, Revision 46, <https://www.kb.cert.org/vuls/id/247371>, 1 Dec. 2001.
- [14] IDA Pro: Disassembler and Debugger, <http://www.hex-rays.com/idadpro/>
- [15] Bil Lewis, Omniscient Debugger: <http://www.lambdacs.com/debugger/>
- [16] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, "Exploiting self-modification mechanism for program protection," *Proc. 27th IEEE Computer Software and Applications Conference*, pp. 170-179, Nov. 2003.
- [17] M. LaDue, "The Maginot license: Failed approaches to licensing Java software over the Internet," <http://www.geocities.com/securejavaapplets/maginot.html>, 1997.
- [18] H. E. Link, and W. D. Neumann, "Clarifying obfuscation: improving the security of white-box encoding," *Cryptology ePrint Archive*, Report 2004/025, International Association for Cryptologic Research, 2004.
- [19] S. Loureiro, and R. Molva, "Function hiding based on error correcting codes," *Proc. International Workshop on Cryptographic Techniques and Electronic Commerce (CRYPTEC99)*, pp. 92-98, July 1999.
- [20] M. Mambo, T. Murayama, and E. Okamoto, "A tentative approach to constructing tamper-resistant software," *Proc. 1997 New Security Paradigm Workshop*, pp. 23-33, Sep. 1997.
- [21] J. Meyer and T. Downing, "Java Virtual Machine," O'Reilly & Associates, Inc., 1997.
- [22] J. Meyer, "D-Java," <http://mrl.nyu.edu/~meyer/jvm/djava/>
- [23] J. Meyer, "Jasmin Home Page," <http://jasmin.sourceforge.net/>
- [24] A. Monden, A. Monsifrot, and C. Thomborson, "Tamper-resistant software system based on a finite state machine," *IEICE Trans. on Fundamentals*, Vol.E88-A, No.1, pp.112-122, Jan. 2005.
- [25] 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).
- [26] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *IEICE Trans. Fundamentals*, Vol.E86-A, No.1, pp.176-186, 2003.
- [27] J. Patarin, and L. Goubin, "Secret key cryptographic process for protecting a computer system against attacks by physical analysis," *United States Patent* 6,658,569, Filed 17 June 1999, Issued 2 Dec. 2003.
- [28] A. Patrizio, "Why the DVD hack was a cinch," *Wired News*, Nov. 1999, <http://www.wired.com/science/discoveries/news/1999/11/32263>
- [29] T. Sander, and C. Tschudin, "Protecting mobile agents from malicious hosts," *Mobile Agents and Security, Lecture Notes in Computer Science*, Vol. 1419, pp. 44-60, 1998.
- [30] Soot: A Java optimization framework, <http://www.sable.mcgill.ca/soot/>
- [31] H. Tamada, "AddTracer, Injecting tracers into Java class files for dynamic analysis," <http://se.naist.jp/addtracer/>
- [32] The decompilation Wiki of Program-Transformation.Org, <http://www.program-transformation.org/Transform/DeCompilation>
- [33] P. M. Tyma, "Method for renaming identifiers of a computer program," *United States Patent* 6,102,966, Assignee: PreEmptive Solutions, Inc., Aug. 2000.
- [34] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl: *Fault Tree Handbook*, Tech. Rep. NUREG-0492, U.S. Nuclear Regulatory Commission, 1981.
- [35] C. Wang, J. Hill, J. Knight, and J. Davidson, "Protection of software-based survivability mechanisms," *Proc. International Conference of Dependable Systems and Networks*, pp. 193-202, July 2001.
- [36] H. Yamauchi, "The evaluation for tamper-resistance of programs based on the instruction sequence differential attack," *Master's Thesis, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0351135*, Feb. 2005 (in Japanese).
- [37] H. Yamauchi, Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, "Software Obfuscation from Crackers' Viewpoint," *Proc. the IASTED International Conference on Advances in Computer Science and Technology*, pp. 286-291, Jan. 2006.



**Hiroki Yamauchi** received the B.E. degree in Electrical Engineering from Doshisha University in 2003, M.E. degree in Information Science Engineering from Nara Institute of Science and Technology of Technology in 2005. He is currently a PhD candidate in Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His research interests include software security

and software requirement. He is a student member of IEEE and IPSJ.



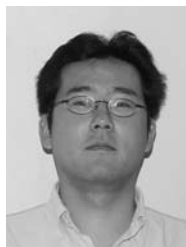
**Akito Monden** received the BE degree (1994) in Electrical Engineering from Nagoya University, Japan, and the ME degree (1996) and DE degree (1998) in Information Science from Nara Institute of Science and Technology, Japan. He was honorary research fellow at the University of Auckland, New Zealand, from June 2003 to March 2004. He is currently Associate Professor at Nara Institute of

Science and Technology. His research interests include software security, software measurement, and human computer interaction. He is a member of the IEEE, ACM, IEICE, IPSJ, JSSST and JSiSE.



**Masahide Nakamura** received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. From 2002 to 2007, he worked for the Graduate School of

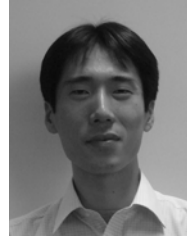
Information Science at Nara Institute of Science and Technology, Japan. He is currently an associate professor in the Graduate School of Engineering at Kobe University. His research interests include the service-oriented architecture, Web services, the feature interaction problem, V&V techniques and software security. He is a member of IEEE and IEICE.



**Haruaki Tamada** received the BE and ME in Information and Communication Engineering from Kyoto Sangyo University, Japan in 1999, 2001. He received DE degree in Information Science from Nara Institute of Science and Technology, Japan in 2006. From 2006 to 2008, he worked for the Graduate School of Information Science at Nara Institute of

Science and Technology, Japan. He is currently an assistant professor in Faculty of Computer Science and Engineering, Kyoto Sangyo University, Japan. His research interests include

software security, software measurement. He is a member of the IEICE, IPSJ and IEEE.



**Yuichiro Kanzaki** received the B.E. degree in computer and systems engineering from Kobe University, Japan in 2001, and the M.E. degree in Information Science engineering from Nara Institute of Science and Technology in 2003. He received DE degree in Information Science from Nara Institute of Science and Technology, Japan in 2006.

He is currently an assistant professor in Department of Information and Computer Sciences, Kumamoto National College of Technology, Japan. His research interests include software protection, software process security and program comprehension. He is a member of the IEICE, IPSJ and IEEE.



**Ken-ichi Matsumoto** received the B.E., M.E., and PhD degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute Science and Technology, Japan. His research interests include software measurement and

software process. He is a senior member of the IEEE, and a member of the ACM and IPSJ.

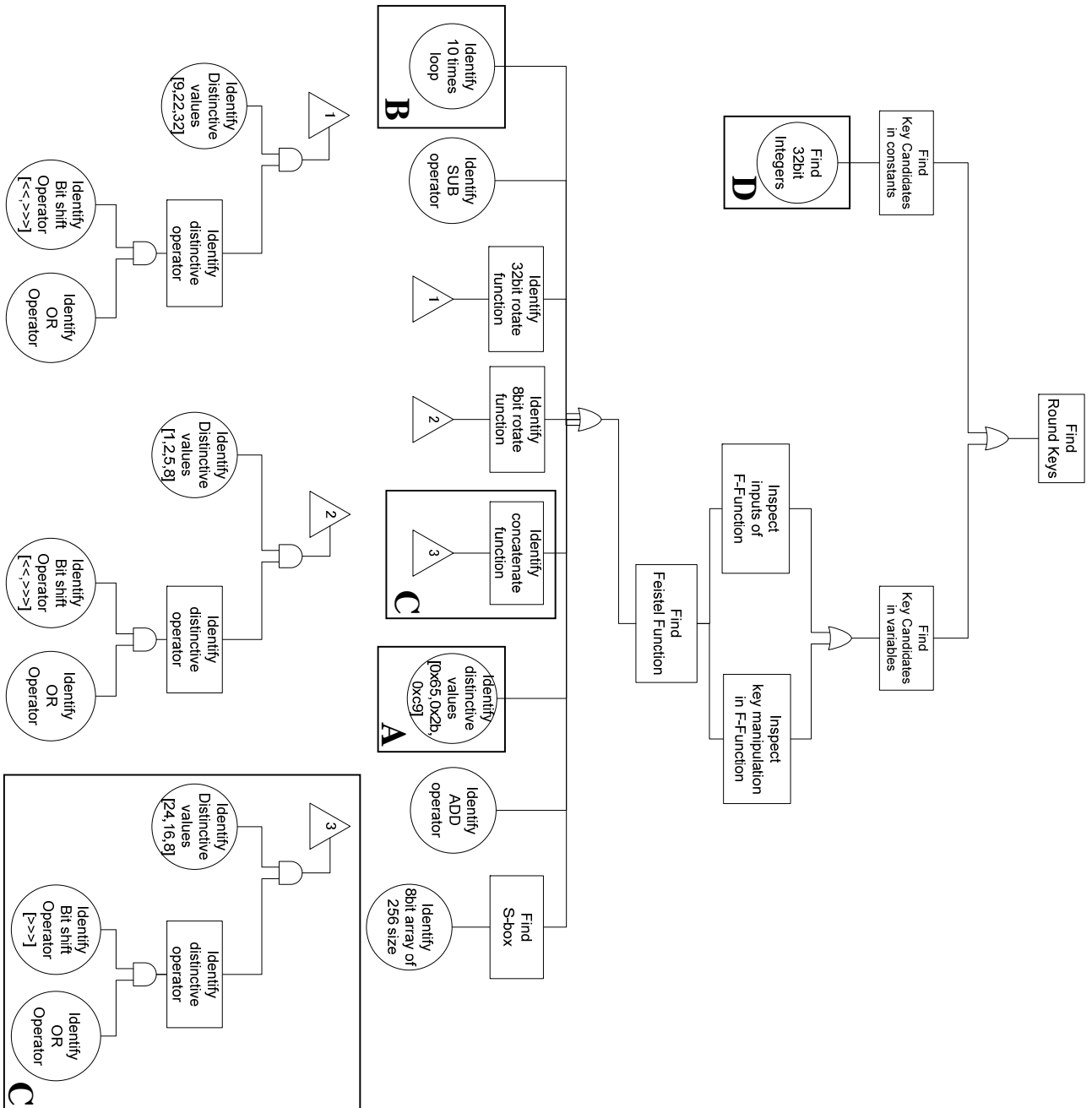


Fig. 7 Goal tree of C2 cipher

```

/* C2 decryption in ECB (Electronic Code Book)
mode: */
public static void c2_d(int inout[]) {
    int L, R, t, round, i;
    int ktmpa, ktmpb, ktmpc, ktmpd;

    /* Round Keys */
    int sk[] = {
        0x789ac6ee, 0x79bc3398,
        0x48d15d62, 0xb3c4da86,
        0xabcde483, 0xc248048f,
        0xfda00b6f, 0xfd600e69,
        0xfe140e66, 0xffee0585
    };

    /* Input Conversion */
    L = inout[0]; R = inout[1];

    for(round=MaxRound-1;round>=0;round--) {
        /* Feistel network */
        L -= F(R, sk[round]);
        t = L; L = R; R = t; // swap
    }
    t = L; L = R; R = t; // swap cancel

    /* Output */
    inout[0] = L; inout[1] = R;
    return;
}

/* F is the Feistel round function: */
public static int F(int data, int key) {
    int t;
    byte v[] = new byte[4];
    byte u;

    /* Key Inersion */
    t = data + key;

    /* Secret Constant */
    v[3] = (byte)((t >>> 24) & 0xff);
    v[2] = (byte)((t >>> 16) & 0xff);
    v[1] = (byte)((t >>> 8) & 0xff);
    v[0] = SecretConstant[t&0xff];

    u = (byte)((v[0]&0xff) ^ 0x65);
    v[1] ^= lrot8(u&0xff, 1);
    u = (byte)((v[0]&0xff) ^ 0x2b);
    v[2] ^= lrot8(u&0xff, 5);
    u = (byte)((v[0]&0xff) ^ 0xc9);
    v[3] ^= lrot8(u&0xff, 2);

    /* Concatenate & Rotate */
    t = (int)v[3] << 24 | (int)v[2] << 16 |
        (int)v[1] << 8 | (int)v[0];
    t ^= lrot32(t,9) ^ lrot32(t,22);
    return t;
}

/* Logical left rotate */
public static byte lrot8(int x, int n) {
    return (byte)(( x << n) | ( x >>> (8-n) ));
}

public static int lrot32(int x, int n) {
    return ( ( x << n) | ( x >>> (32-n) ));
}

```

Fig. 8 Source code of C2 cipher (before obfuscation)

```

/* C2 decryption in ECB (Electronic Code
Book) mode: */
public static void c2_d(int inout[]) {
    int L, R, t;

    /* Round Keys */
    long enc_key[]={
        0x3ffb81617L, 0x3f850399bL,
        0x3f58039a7L, 0x3f6802dbfL,
        0x30920123fL, 0x2af37920fL,
        0x2cf136a1bL, 0x12345758bL,
        0x1e6f0ce63L, 0x1e26b1bbbL
    };

    /* Input Conversion */
    L = inout[0]; R = inout[1];

    L -= foo(R, enc_key[0]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[1]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[2]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[3]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[4]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[5]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[6]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[7]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[8]);
    t = L; L = R; R = t; // swap
    L -= foo(R, enc_key[9]);

    /* Output */
    inout[0] = L; inout[1] = R;
    return;
}

/* foo is the Feistel round function: */
public static int foo(int data, long enc_key) {
    long t, t2, t3;

    int tt;
    byte v[] = new byte[4];
    byte u;

    /* Key Inserion */
    t = data * 4 + enc_key ;

    /* Secret Constant */
    t = (t - 3) / 4;

    t3 = t / 256; t3 /= 256; t3 /= 256;
    t2 = t / 256; t2 /= 256;
    v[3] = (byte)(t3&0xff);
    v[2] = (byte)(t2&0xff);
    v[1] = (byte)((t/256)&0xff);
    v[0] = SecretConstant[(int)t&0xff];

    u = (byte)(v[0] ^ 0x21);
    u = (byte)(u ^ 0x44);
    v[1] ^= lrot8(u,1);
    u = (byte)(v[0] ^ 0x28);
    u = (byte)(u ^ 0x03);
    v[2] ^= lrot8(u,5);
    u = (byte)(v[0] ^ 0x41);
    u = (byte)(u ^ 0x88);
    v[3] ^= lrot8(u,2);

    /* Concatenation & Rotation */
    tt = ~(~((int)v[3] * 16777216) & ~((int)v[2]
    * 65536)) & ~(~((int)v[1] * 256) & ~((int)v[0]));
    tt ^= lrot32(tt,9) ^ lrot32(tt,22);
    return tt;
}

/* Logical left rotate */
public static byte lrot8(int x, int n) {
    return (byte)( ( x << n ) | ( x >>> (8-n) ) );
}

public static int lrot32(int x, int n) {
    return ( ( x << n ) | ( x >>> (32-n) ) );
}

```

Fig. 9 Source code of C2 cipher (after obfuscation)