Vol. 47 No. 8 IPSJ Journal Aug. 2006

Regular Paper

An Obfuscation Scheme Using Affine Transformation and Its Implementation

Kazuhide Fukushima,[†] Shinsaku Kiyomoto[†]
and Toshiaki Tanaka[†]

Program analysis techniques have improved steadily over the past several decades, and these techniques have made algorithms and secret data contained in programs susceptible to discovery. *Obfuscation* is a technique to protect against this threat. Obfuscation schemes that encode variables have the potential to hide both algorithms and secret data. We define five types of attack — *data-dependency attacks*, *dynamic attacks*, *instruction-guessing attacks*, *numerical attacks*, and *brute force attacks* — that can be launched against existing obfuscation schemes. We then propose an obfuscation scheme which encodes variables in a code using an affine transformation. Our scheme is more secure than that of Sato, et al. because it can protect against *dependency attacks*, *instruction-guessing attacks*, and *numerical attacks*. We describe the implementation of our scheme as an obfuscation tool for C/C++ code.

1. Introduction

1.1 Background

Program analysis techniques have been steadily improved over several decades, as evidenced by the development of tools such as debuggers and decompilers. Algorithms and secret data contained in programs may be susceptible to discovery through the use of these tools. For example, it was reported that Apple's jukebox program "iTunes" was analyzed and their copyright management technology "FairPlay" cracked[10]. In this case, the attacker analyzed the program and exposed the method used to generate the secret key for encrypting music. Even though program development may require a great deal of labor, it is relatively easy for attackers to steal the algorithms and secret data contained in a program. Therefore, techniques that make programs difficult to analyze are desirable for copyright protection and to maintain the secrecy of data.

The term *obfuscation* refers to such techniques. Obfuscation schemes transform an original program (code or a binary file) into an obfuscated program that is more difficult to analyze while preserving its functionality. Here, we define functionality as preserved if the obfuscated program gives the same output as that of the original program when both programs have common input. Obfuscation prevents substantial analysis of a program if the cost of analyzing the obfuscated program exceeds the value of the original program.

1.2 Related Work

1.2.1 Existing Schemes

Many obfuscation schemes have been proposed.

Monden, et al. proposed an automatic obfuscation scheme for C code that contains loops[8], and Gannod proposed a scheme that obfuscates `while` loops, `do while` loops, and `for` loops[7]. Collberg, et al. proposed a scheme that inserts dummy instructions using conditional branching[3]. Chan, et al. proposed a scheme that modifies identifiers contained in Java byte code in order to protect the code against decompilation[2]. Sosonkin, et al. proposed a scheme that changes the structure of classes in a Java code by merging and dividing them[13]. Obfuscation schemes with a theoretical basis have also been proposed. Wang, et al. showed that the problem of determining to which address a program branches statically is NP-hard, and proposed an obfuscation scheme for C code by indirect reference using arrays and pointers[14]. Ogiso, et al. showed that the problem of determining to which address a function pointer points statically is NP-hard, and proposed an obfuscation scheme using function pointers that obfuscate function calls[9]. Sakabe, et al. showed that the problem of determining a points-to in a code with method overloading and classes which implement interfaces is NP-hard, and proposed an obfuscation scheme for Java codes.

Collberg, et al.[4] and Sato, et al.[12] both proposed obfuscation schemes that encode variables. One of the schemes proposed by Coll-

† KDDI R&D Laboratories Inc.

berg, et al., *change encoding*, encodes the control variables of loops with a linear function. However, they did not show a concrete procedure for encoding variables and instructions. Sato, et al. proposed a systematic obfuscation scheme that encodes variables and instructions[12].

Most existing schemes aim at obfuscating the algorithm in the target program. Obfuscation schemes[3),7)~9),11),13),14)] change the structure of a program. However, these schemes do not explain how to protect secret data. On the other hand, encoding schemes transform data according to certain rules, and we can use these schemes to protect secret data. Furthermore, following transformation of the variables, the operations between them are also transformed. Therefore, an obfuscation scheme using encoding potentially can hide both algorithms and secret data.

1.2.2 Approach of Sato, et al.

The scheme proposed by Sato, et al. encodes a variable x using a linear transformation $C(a,b) : x \mapsto ax + b \quad (a \neq 0 \text{ and } a, b \in \mathbb{Z})$. X and Y denote the encoded variables obtained from the variables x and y, respectively—that is, $X = ax + b$ and $Y = ay + b$. The transformation is called the *encoding rule* $C(a,b)$ for variable x. They apply the transformation rules $R_1(a,b)$, $R_2(a,b)$, $\ldots$, $R_{10}(a,b)$ (**Table 1**) to arithmetic instructions in code according to the priority of operators. As a result, they obtain code obfuscated by the coding rule $C(a,b)$. These rules $R_1(a,b)$, $R_2(a,b)$, $\ldots$, $R_{10}(a,b)$ are called the *operator-transformation rules* $R(a,b)$ corresponding to the coding rules $C(a,b)$. All the operator-transformation rules $R(a,b)$ are shown in Table 1. Each coding rule $C(a,b)$ corresponds to exactly one operator transformation rule $R(a,b)$. That is, when a and b of the code rule $C(a,b)$ are fixed, the operation-transformation rule $R(a,b)$ is determined. Finally, they apply the *decode rule* $D(a,b) : ax + b \mapsto x$ to the result. The plain result is obtained through the decoding rule $D(a,b)$.

1.3 Our Contribution

First, we define five types of attack: *data-dependency attacks*, *dynamic attacks*, *instruction-guessing attacks*, *numerical attacks*, and *brute force attacks.*

Each can be applied to a code obfuscated by Sato, et al.'s scheme. This scheme is vulnerable to these attacks because the original variable and the encoded variable have a one-to-one correspondence. We then propose an obfuscation scheme that simultaneously encodes multiple variables in a code through an affine transformation. Our scheme is more secure than that of Sato, et al. because it can protect against *data-dependency attacks*, *instruction-guessing attacks*, and *numerical attacks.*

We have quantitatively evaluated the efficiency and the *complexity of variable dependences* in obfuscated codes. In our toy example, the number of instructions in our obfuscated assembly code was 3.67 times the original number. However, we can improve the execution efficiency by applying our scheme only to parts where the highest level of security is required. Moreover, the *complexity of variable dependences* in code obfuscated by our scheme is greatly increased, while that in code obfuscated by Sato's scheme does not increase.

We have implemented our scheme as an obfuscation tool for C/C++ code.

Table 1 All the operator-transformation rules $R(a,b)$.

$R_1(a,b) : X + Y \to X + Y - b$
$R_2(a,b) : X - Y \to X - Y + b$
$R_3(a,b) : X * Y \to (XY - b(X + Y - b - a))/a$
$R_4(a,b) : X/Y \to (aX + bY - b(b + a))/(Y - b)$
$R_5(a,b) : X + y \to X + ay$
$R_6(a,b) : X - y \to X - ay$
$R_7(a,b) : y - X \to ay - X + 2b$
$R_8(a,b) : y * X \to yX - by + b$
$R_9(a,b) : X/y \to X/y - b/y + b$
$R_{10}(a,b) : y/X \to (a^2y)/(X - b) + b$

2. Attacks

Some code includes secret data needed for the program itself; e.g., keys for a copyright management program. Attackers can obtain secret data by investigating algorithms in cases where the data is divided or transformed using certain functions.

2.1 Attacks on Obfuscated Code

We classify attacks on obfuscated code into three types.

- Attacks that obtain secret algorithms and secret data directly from obfuscated code
- Attacks that guess the original instructions and variables in the original code from the obfuscated code
- Attacks that deobfuscate the obfuscated code by finding some secret keys (Note that the secret key differs from the “secret data” defined above. The former is not contained

in the code while the latter is.)
The first and second types of attack can succeed if attackers obtain the obfuscated code. However, the third type of attack can succeed only if an obfuscation scheme uses a secret key and the attackers know that the scheme is being applied.

Furthermore, attackers must investigate all of the code in the first and second types of attack, and the cost of these attacks rises as the code size increases. However, the cost of the third type of attack does not increase with the code size since the attackers do not directly investigate the obfuscated code.

We classify *data-dependency attacks* and *dynamic attacks* into the first type; *instruction-guessing attacks* into the second type; and *numerical attacks* and *brute force attacks* into the third type.

2.2 Detailed Description of Attacks

The following provides detailed descriptions of the five kinds of attack.

(1) Data-Dependency Attack

A *data-dependency attack* succeeds if an attacker can obtain the value of a variable from an obfuscated code using the dependencies between variables.

Obfuscated codes include many variables, and attackers want to know the values of these variables. However, a value is rarely assigned as an immediate value. In most cases, it depends on the previous value of the variable itself or values of other variables. Thus, attackers must obtain the values of variables using the dependence between variables; i.e., the relations of references and assignments. These values can be used as clues to analyze algorithms and secret data.

For example, attackers may derive algorithms used in obfuscated code by investigating the value of variables which hold the output. Additionally, some secret data may be divided into multiple variables or may be transformed by some functions. In this case, attackers can obtain the secret data through this sort of attack.

(2) Dynamic Attack

A *dynamic attack* succeeds if an attacker can obtain some information concerning the obfuscated code.

This information can then be used as a clue to analyze an algorithm and secret data.

For example, attackers add instructions to output the values of variables to obfuscated code and then execute the code. This allows them to obtain the value of arbitrary variables at any point. They can then obtain the algorithm and secret data through the method used in a *data-dependency attack*.

(3) Instruction-Guessing Attack

An *instruction-guessing attack* succeeds if an attacker can guess the instructions in the original code from those in obfuscated code.

The attacker can derive which variables appear and how the variables are used in the instructions of the original code. This information can be used as a clue to analyze an algorithm and secret data.

For example, the attacker may determine an algorithm by using guessed instructions. Additionally, the attacker may find which data is used as an input to an encryption function; that is, which data is secret data.

(4) Numerical Attack

A *numerical attack* succeeds if an attacker can deobfuscate the obfuscated code by using numerical relations between the obfuscated data and the original data.

This attack can directly deobfuscate the obfuscated code, and attackers can obtain algorithms and secret data from unprotected instructions and variables. It can be applied to obfuscation schemes in which the original data and obfuscated data are numerically related.

(5) Brute Force Attack

A *brute force attack* succeeds if an attacker can obtain the secret key used for obfuscation by trying all possible keys.

If attackers can find the secret key, they can obtain the original code. They can then obtain algorithms and secret data from unprotected instructions and variables.

2.3 Resistance of Existing Schemes

Here, we discuss how resistant existing schemes are to the five kinds of attack.

Some obfuscation schemes [3),4),9),11)~14)] are effective against *data-dependency attacks*. Some of these schemes change the number of variables [3),9),11),13),14)], and others change the instruction semantics [4),12)]. The obfuscation schemes that obfuscate only the control structures in code [7),8)] are the least effective against this sort of attack.

Likewise, some obfuscation schemes[3),4),9),11),12),14)] are effective against *instruction-guessing attacks*. Some change the number of instructions by introducing dummy instructions [3),9),11),14)] and others change the semantics of instructions [4),12)]. However, certain schemes [3),7)~9),11),13),14)] are not very effective against this sort of attack. The schemes proposed by Monden and Gannod [7),8)] do not change instructions while changing the control structures, and that of Sosonkin [13)] changes only the data structures.

None of these schemes are effective against *dynamic attacks*, since every attacker who has an obfuscated code can succeed in such an attack.

Numerical attacks and *brute force attacks* can succeed against only the schemes of Collberg and Sato [4),12)]. These two schemes use a numerical function to encode variables. Additionally, two coefficients in the function are used as a key.

However, these two schemes are superior to other schemes in two respects. First, they change the instruction semantics, and automatic attacks using pattern matching are difficult to apply. Second, transformation has many variations since it uses a secret key.

3. Problems with the Scheme of Sato, et al.

Sato, et al.'s scheme is somewhat effective against *data-dependency attacks* and *instruction-guessing attacks*. However, this scheme encodes single variables with a linear function—that is, there exists exactly one encoded variable for each original variable. Thus, the effectiveness is limited. Furthermore, their scheme cannot protect against *numerical attacks*, *dynamic attacks*, or *brute force attacks*.

(1) Data-Dependency Attack

Sato, et al.'s scheme cannot fully protect against this form of attack. Their scheme encodes a variable into exactly one encoded variable, so it does not obfuscate the dependence between variables though some expressions become more complicated.

Here, we provide two simple examples.

First, we discuss protection for algorithms. We assume that attackers want to obtain the value of Y after the `while` loop is executed once in the obfuscated code (**Fig. 1**), and they use the value as a clue to analyze algorithms included in the code. In this case, the attackers must obtain the previous values of variables TMP and Y since Y is last updated by the instruction `Y = -(TMP+2)+Y;` in the 11th line. $Y = -1$ is obtained from the fourth line, and TMP is updated by the instruction `TMP = (X-2)/2;` in the ninth line. The attackers must obtain the previous value of X, and can find that the value is 2 from the instruction `X = 2;` in the third line. As a consequence, they must obtain the values of variables $TMP, Y, andX$ to obtain the variable Y in the obfuscated code. On the other hand, they must obtain the value of variables $tmp, y, andx$ to obtain the corresponding variable y also in the original code (**Fig. 2**).

```
 1: int main(void){
 2:  int X, Y, TMP, C, l;
 3:  X = 2;
 4:  Y = -1;
 5:  TMP = -1;
 6:  C = 1;
 7:  scanf("%d", &l);
 8:  while((C-1)/2 < l){
 9:   TMP = ((X-2)/2)-2;
10:   X = -2*Y+2;
11:   Y = -(TMP+2)+Y;
12:   C = C+2;
13:  }
14:  printf("Fibonacci(%d)=%d\n",l,(X-2)/2);
15:  return 0;
16: }
```

Fig. 1 Code (1) obfuscated by Sato, et al.'s scheme.

```
 1: int main(void){
 2:  int x, y, tmp, c, l;
 3:  x = 0;
 4:  y = 1;
 5:  tmp = 1;
 6:  c = 0;
 7:  scanf("%d",&l);
 8:  while(c < l){
 9:   tmp = x;
10:   x = y;
11:   y = tmp+y;
12:   c = c+1;
13: }
14: printf("f Fibonacci(%d) = %d\n", l, x);
15: return 0;
16: }
```

Fig. 2 Original code (1).

Second, we discuss protection for secret data. We assume that attackers want to obtain secret data used in the `if` statement in the seventh line in the obfuscated code (**Fig. 3**). In the obfuscated code, the value of the variable k is divided into the variables x and y. The value of k is secret data since it affects whether function `LicenseOK()` is executed. The attackers can obtain the value of K by determining the values of variables X and Y, also in the obfuscated code. On the other hand, they must

```
 1: int LicenseCheck(void){
 2:  int x,y,k,i;
 3:  x = 3;
 4:  y = 4;
 5:  k = x+y;
 6:  scanf("%d",&i);
 7:  if (i == k){
 8:   LicenseOK();
 9:   return 0;
10:  }
11:  return -1;
12: }
```

Fig. 3 Original code (2).

```
 1: int LicenseCheck(void){
 2:  int X,Y,K,i;
 3:  X = 7;
 4:  Y = -1;
 5:  K = -(3*X+6*Y+9)/2;
 6:  scanf("%d", &i);
 7:  if (i == (9-K)/3){
 8:   LicenseOK();
 9:   return 0;
10:  }
11:  return -1;
12: }
```

Fig. 4 Code (2) obfuscated by Sato, et al.'s scheme.

determine the values of the two variables to obtain the value of k, also in the original code (**Fig. 4**).

Thus, attackers can obtain the value of a variable in obfuscated code if they can obtain as many of the previous values of a variable as exist in the original code, though some instructions become more complicated.

(2) Dynamic Attack

Sato's scheme cannot protect against this form of attack. For example, attackers can obtain values of arbitrary variables at any point by inserting `printf` at the point shown in Fig. 1 and Fig. 4. These values are used as clues to analyze algorithms and secret data with the method used in the *data-dependency attack.*

(3) Instruction-Guessing Attack

Sato's scheme cannot fully protect against this form of attack. There exists exactly one original variable for each encoded variable, and the two variables have a one-to-one correspondence. An attacker can guess the original instruction using these correspondences.

We again provide two simple examples.

First, we discuss protection for algorithms. We assume that attackers want to determine the structure of the original instructions in the obfuscated code (Fig. 1). The instruction `Y = -(TMP+2)+Y;` in the 11th line assigns a value to Y that is computed from the encoded variables TMP and Y. The attackers can guess that the structure of the original instruction is `y=f(tmp,y);`, where y and tmp are the original variables corresponding to Y and TMP, respectively, and f is some numerical function. In fact, the original instruction is `y = tmp+y` (see the 11th line of Fig. 2).

Second, we discuss protection for secret data. We assume that attackers want to determine the condition where the function `LicenseOK()` is executed in the obfuscated code (Fig. 4). They will focus on the `if` statement in the seventh line of the code. Only the variable K is compared with the input. They can guess that only one variable is likewise compared with the input in the original code and this data is secret. Finally, they can determine the condition by finding the value of K.

Sato's obfuscation scheme changes the instruction semantics, but does not change the number of variables that appear in each instruction. Thus, attackers can guess the instructions in the original code.

(4) Numerical Attack

Sato's scheme cannot protect against this form of attack. Attackers can decode a variable if they can obtain exactly one encoding rule for the variable, and they can obtain the algorithm and secret data from unprotected instructions and variables.

Here, we show the decoding process:

Step 1 Obtain the Decoding Rule

Attackers obtain decoding rule $D(a, b)$ for variable x. The decoding rule can be obtained at the points where the result is decoded.

Step 2 Find the Encoding Rule

The attackers find encoding rule $C(a, b)$ by calculating the inverse function of decoding rule $D(a, b)$.

Step 3 Decode the Variable

The attackers decode encoded variable X by executing their obfuscation scheme and interchanging the encoding rule and the decoding rule. That is, decoding rule $D(a, b)$ is used as encoding rule $C(1/a, -b/a)$, and encoding rule $C(a, b)$ is used as decoding rule $D(1/a, -b/a)$.

The following is a simple example. We assume attackers want to decode variable X in

```
 1: int main(void){
 2:  int x, Y, TMP, C, l;
 3:  x = 0;
 4:  Y = -1;
 5:  TMP = -1;
 6:  C = 1;
 7:  scanf("%d", &l);
 8:  while((C-1)/2 < l){
 9:   TMP = x-2;
10:   x = -Y;
11:   Y = -(TMP+2)+Y;
12:   C = C+2;
13:  }
14:  printf("Fibonacci(%d) = %d\n", l, x);
15: }
```

Fig. 5 Code (1) in which variable x is decoded.

the obfuscated code (Fig. 1). First, they focus on the 14th line. They can guess that $D(2,2) = (X-2)/2$ is the decoding rule for X since the (decoded) result is output in this line. They can then decode variable x using the encoding rule and the decoding rules. The resultant code is shown in **Fig. 5**. Furthermore, they may guess $D(2,1) = (C-1)/2$ is the decoding rule for C since the variable is compared with the input data in the eighth line.

(5) Brute Force Attack

Their scheme cannot fully protect against this form of attack. Attackers can decode a variable with the method used in the *numerical attack* if they can correctly guess the decoding rule for the variables.

An attacker must specify two integers used in the encoding transformation. Thus, the cost of this attack is N^2, where the number of possibilities for each element is N. This attack is difficult when N is large. However, the calculable range of a program may be greatly restricted in this case. Thus, there is a trade-off between the security and the functionality of an obfuscated code.

4. Our Scheme

We propose an obfuscation scheme that simultaneously encodes multiple variables in a code with an affine transformation. In our scheme, there is no one-to-one correspondence between each original variable and each encoded variable since the data a variable holds is distributed among multiple encoded variables. Therefore, our scheme can protect against the *data-dependency attacks*, *instruction-guessing attacks*, and *numerical attacks* defined in Section 2.2.

4.1 Encoding of Variables

We encode n variables $x_1, x_2, \ldots, x_n$ to m variables $X_1, X_2, \ldots, X_m$ with an $m \times n$ matrix $A = (a_{i,j})$ and an m-dimensional vector $\boldsymbol{b} = (b_1 b_2 \ldots b_m)^T$.

Here, we assume that n and m are positive integers such that $m \geq n$, $a_{i,j}, b_j \in \mathbb{Z}$, and $\mathrm{rank}(A) = n$. Thus, the affine transformation $C(A, \boldsymbol{b}) : \boldsymbol{x} \mapsto A\boldsymbol{x} + \boldsymbol{b}$ is defined by

$$\boldsymbol{X} = A\boldsymbol{x} + \boldsymbol{b}, \qquad (1)$$

where $\boldsymbol{x} = (x_1 x_2 \ldots x_n)^T$ and $\boldsymbol{X} = (X_1 X_2 \ldots X_m)^T$. We call the transformation the *encoding transformation* $C(A, \boldsymbol{b})$. We then take out each line of equation (1) and obtain

$$X_i = \sum_{j=1}^{n} a_{ij} x_j + b_j \ \ (1 \leq i \leq m). \qquad (2)$$

We call each equation of (2) an *encoding rule.* We can replace instructions assigning values to the variables $x_1, x_2, \ldots, x_n$ with instructions assigning other values to the encoded variables $X_1, X_2, \ldots, X_m$ using the encoding rule.

Note that encoding rules are simultaneous equations, so we can find the original variables $x_1, x_2, \ldots, x_n$ from the encoded equations. Here, the equation

$$\mathrm{rank}A = \mathrm{rank}(A|\boldsymbol{X} - \boldsymbol{b}) = n$$

must be satisfied to find unknown n variables uniquely from the m equations. Thus, the encoded variables $X_1, X_2, \ldots, X_m$ must satisfy the $m-n$ non-trivial relational equations

$$r_t(X_1, X_2, \ldots, X_m) = 0 \ \ (1 \leq t \leq m-n).$$

The original variables $x_1, x_2, \ldots, x_n$ are then given by

$$x_i = f_i(X_1, X_2, \ldots, X_m) + \sum_{t=1}^{m-n} c_{i,t} r_t(X_1, X_2 \ldots, X_m) \ \ (1 \leq i \leq n) \qquad (3)$$

using the encoded variables $X_1, X_2, \ldots, X_m$, where $c_{i,t}$ are arbitrary-constants and f_i are linear functions of $X_1, X_2, \ldots, X_m$. We call each equation of (3) a *decoding rule.* Each equation defines the decoding transformation $D(A, \boldsymbol{b}) : A\boldsymbol{x} + \boldsymbol{b} \mapsto \boldsymbol{x}$. We can replace references to the variables $x_1, x_2, \ldots, x_n$ with references to the variables $X_1, X_2, \ldots, X_m$ using the decoding rules. The encoding rules contain the arbitrary constants $c_{i,t}$. However, we can uniquely find $x_1, x_2, \ldots, x_n$ using the decoding rules because the product of $c_{i,t}$ and the non-trivial relational equations $r_t(X_1, X_2, \ldots, X_m)$ is 0. We can arbitrarily change the coefficients on the variables $X_1, X_2, \ldots, X_m$ in the decoding rules by changing the arbitrary constants $c_{i,t}$.

4.2 Applying the Scheme to Codes

Our scheme can be applied to codes written in high-level languages, to assembly code, and to machine languages for various processors. We obfuscate a code in the following way:

Step 1 Select Target Variables
Select n integer variables $x_1, x_2, \ldots, x_n$ arbitrarily from the code (with the exception of the variables storing the input).

Step 2 Generate a Matrix and a Vector
Generate an $m \times n$ matrix $A = (a_{i,j})$ and an m-dimensional vector $\boldsymbol{b} = (b_1 b_2 \ldots b_m)^T$, where $m \geq n$, $a_{i,j}, b_j \in \mathbb{Z}$, and $\mathrm{rank}(A) = n$.

Step 3 Define the Encoding Rules
Define the encoding rules

$$X_i = \sum_{j=1}^{n} a_{ij} x_j + b_j \ \ (1 \leq i \leq m),$$

where variables $X_1, X_2, \ldots, X_m$ are the encoded variables to be used in the obfuscated code, using matrix A and vector $\boldsymbol{b}$.

Step 4 Derive the Decoding Rules
Consider the encoding rules defined in Step 3 to be simultaneous equations, and find $x_1, x_2, \ldots, x_n$ to obtain the decoding rules

$$x_i = f_i(X_1, X_2, \ldots, X_m) \ \ (1 \leq i \leq n).$$

Step 5 Encode the Variables
Replace all the variables $x_1, x_2, \ldots, x_n$ used in the code with the encoded variables $X_1, X_2, \ldots, X_m$ according to the following steps:

(a) Encode the Assignment Instructions
Replace assignment instructions to the variables $x_1, x_2, \ldots, x_n$ with assignment instructions to the encoded variables $X_1, X_2, \ldots, X_m$. Generally, the assignment instruction $x_i \leftarrow v$ is replaced with the instruction

$$\boldsymbol{X} \leftarrow A\boldsymbol{x}|_{x_i \leftarrow v} + \boldsymbol{b},$$

where $\boldsymbol{x}|_{x_i \leftarrow v} = (x_1 x_2 \ldots x_{i-1} v x_{i+1} \ldots x_n)^T$. All the encoded variables $X_1, X_2, \ldots, X_m$ are assigned when this assignment instruction is executed.

(b) Encode References to Variables
Replace references to variables $x_1, x_2, \ldots, x_n$ in the code with the decoding rules $x_i(X_1, X_2, \ldots, X_m)$ derived in Step 4. Choose arbitrary natural numbers for the constants $c_{i,t}$ contained in the decoding rules.

Step 6 Modifications
Give initial values to the encoded variables and merge consecutive assignment instructions as post-processing.

(a) Add Initialize Instructions
Add assignment instructions which give initial values to the encoded variables $X_1, X_2, \ldots, X_m$. These initial values must satisfy the non-trivial relational equations

$$r_t(X_1, X_2, \ldots, X_m) \ \ (1 \leq t \leq m-n).$$

(b) Merge instructions
Merge the consecutive assignment instructions produced in Step 5 (b). For example, the following two consecutive assignment instructions

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \leftarrow \begin{pmatrix} 2X_1 - 3X_2 + 5 \\ -X_1 + 4X_2 + 2 \end{pmatrix}$$
$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \leftarrow \begin{pmatrix} X_1 + X_2 - 3 \\ 2X_1 + 3X_2 + 1 \end{pmatrix}$$

can be merged into exactly one assignment instruction

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \leftarrow \begin{pmatrix} X_1 + X_2 + 4 \\ X_1 + 6X_2 + 17 \end{pmatrix}.$$

This procedure reduces the number of instructions to increase execution efficiency.

(c) Decompose instructions
Decompose instructions into successive multiple instructions for exactly one variable. For example, the assigning instruction for the two encoded variables X_1 and X_2

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \leftarrow \begin{pmatrix} 2X_1 - 3X_2 + 5 \\ -X_1 + 4X_2 + 2 \end{pmatrix}$$

is decomposed into the following successive instructions $T_1 \leftarrow X_1$, $T_2 \leftarrow X_2$, $X_1 \leftarrow 2T_1 - 3T_2 + 5$, and $X_2 \leftarrow -T_1 + 4T_2 + 2$. Here, T_1 and T_2 are temporary variables for saving the values of variables X_1 and X_2, respectively. This procedure transforms assigning instructions for multiple variables into those for a unitary variable which can be realized in real code.

We can gradually make the analysis of a code more difficult by repeatedly applying the process from Step 1 to Step 6.

5. Analysis

First, we analyze the security of our scheme. We show that our scheme can protect against the five types of attack we have identified. Then, we analyze the efficiency of code obfuscated by our scheme. The increase in the number of instructions is estimated quantitatively. Finally, we investigate the change in the functionality, the efficiency, and the *complexity of variable dependences* in the analysis of an ac-

```
 1: int main(void){
 2:  int Q003, Q007, Q004, Q008, Q002,
 3:  Q006, Q001, Q005, l;
 4:  Q005 = -1;
 5:  Q006 = 5;
 6:  Q007 = -9;
 7:  Q008 = 2;
 8:  Q001 = 0;
 9:  Q002 = 2;
10:  Q003 = 0;
11:  Q004 = -3;
12:  scanf("%d", &l);
13:  while((((-Q001)+Q002+Q003-2)/2) < l){
14:   Q005 = Q001;
15:   Q006 = Q002;
16:   Q007 = Q003;
17:   Q008 = Q004;
18:   Q001 = (Q005-2*Q006-Q007-2*Q008-2)/2;
19:   Q002 = Q005+Q006+Q007-2;
20:   Q003 = (-3*Q005-2*Q006-Q007-2*Q008+6)/2;
21:   Q004 = (-2*Q005-Q006-2*Q007+2*Q008+4)/2;
22:  }
23:  printf("Fibonacci(%d)=%d\n",l,
24:   (Q001+Q002+Q003-2)/2);
25:  return 0;
26: }
```

Fig. 6 Code (1) obfuscated by our scheme.

```
 1: int LicenseCheck(void){
 2:  int V001,V002,V003,V004,V005,V006,i;
 3:  V004 = 11;
 4:  V005 = -3;
 5:  V006 = 9;
 6:  V001 = V004;
 7:  V002 = V005;
 8:  V003 = V006;
 9:  V004 = V001-V003+5;
10:  V005 = -V002;
11:  V006 = 5;
12:  scanf("%d",&i);
13:  if (i == (3*V004-V005-5*V006+15)/2){
14:   LicenseOK();
15:   return 0;
16:  }
17:  return -1;
18: }
```

Fig. 7 Code (2) obfuscated by our scheme.

tual code.

5.1 Analysis of Security

We analyze the resistance of our obfuscation scheme to the five types of attack defined in Section 2.2. Our scheme provides higher security against *data-dependency attacks*, *instruction-guessing attacks*, and *numerical attacks* than Sato, et al.'s scheme. However, our scheme cannot fully protect against *dynamic attacks* or *brute force attacks*.

(1) Data-Dependency Attack

Our scheme provides higher security against this form of attack

In our scheme, multiple variables are encoded simultaneously. The data that a variable holds is distributed among multiple encoded variables, and the analysis of dependences between variables in an obfuscated code is more difficult than that of dependences in the original one.

We show two examples.

First, we discuss protection for algorithms. We assume an attacker wants to find the value of $Q003$ after a `while` loop is executed once in the obfuscated code (**Fig. 6**) in order to analyze the algorithms. The value of $Q003$ is finally updated by the instruction in the 20th line. This instruction refers to the value of variables $Q005$, $Q006$, $Q007$, and $Q008$. Furthermore, these variables are updated with the values of $Q001$, $Q002$, $Q003$, and $Q004$, respectively, by instructions from the 14th line to the 17th line. Thus, the attacker must obtain the previous values of eight variables to find the desired value. On the other hand, in the original code (Fig. 2), the attacker only has to obtain the value of three variables, tmp, y, and x, to find the value of the corresponding variable y.

Second, we discuss protection for secret data. We assume an attacker wants to find the value that is compared with input i in the 16th line in the obfuscated code (**Fig. 7**). An attacker who can find the secret data will be able to execute the function `LicenseOK()`. However, the attacker must obtain many variables to find the data; in contrast, the attacker can find the secret data k if they obtain two values, x and y, in the original code.

(2) Dynamic Attack

Our scheme cannot fully protect against this form of attack. For example, attackers can obtain values of arbitrary variables at any point by inserting `printf` at the point shown in Fig. 6 and Fig. 7. These values can be used as clues to analyze algorithms and secret data with the method used in the *data-dependency attack*.

However, the number of variables increases when $m > n$. In this case, the number of variables that an attacker must investigate increases, so the attack cost increases.

(3) Instruction-Guessing Attack

Our scheme provides higher security against this form of attack.

In our scheme, a one-to-one correspondence between each original variable and each encoded variable does not exist in the obfuscated code. Furthermore, an assignment instruction to one variable is replaced by multiple instructions that affect all the encoded variables.

First, we discuss protection for algorithms. The instructions from the 14th line to the 21st line in the obfuscated code (Fig. 6) correspond to the instructions from the ninth line to the 12th line in the original code (Fig. 2). The number of instructions and variables that appear in the instructions differ. Thus, attackers cannot guess the original instructions from the instructions in the code.

Second, we discuss protection for secret data. The `if` statement in the 13th line in the obfuscated code (Fig. 7) refers to the variable $V004$, $V005$, and $V006$ to determine whether to execute the function `LicenseOK()`. Attackers cannot guess which variable holds secret data that is used for the decision in the original code, and they must investigate all the values of the variables.

(4) Numerical Attack

Our scheme also provides higher security against this form of attack.

In our scheme, attackers cannot find any encoding rules since it is impossible to find the values of the multiple unknown variables $X_1, X_2, \ldots, X_m$ from just one decoding rule. Thus, this attack cannot be applied to the obfuscated code.

The following provides an example. The result is output through the expression `(Q001+Q002+Q003-2)/2` in the 24th line in the obfuscated code (Fig. 6). Attackers guess that the expression is the decoding rule for the original variable x since the result is output in this line. They might then try to derive the encoding transformation. However, they must find three unknown variables, $Q001$, $Q002$, and $Q003$, from just one equation $(Q001 + Q002 + Q003 - 2)/2 = x$, which is impossible. Thus, attackers must obtain all the decoding rules.

(5) Brute Force Attack

Our scheme cannot fully protect against this form of attack. An attacker can derive the decoding transformation if they can guess the encoding transformation. They can then obtain the original code by executing our scheme and interchanging the encoding transformation and the decoding transformation. That is, by using the decoding transformation as the encoding transformation and the encoding transformation as the decoding transformation.

The attacker must specify all the elements of the matrix and the vector used in the encoding transformation. Thus, the cost of this attack is $N^{m(n+1)}$, where the number of possibilities for each element is N. This attack is difficult when m, n, and N are large. However, the calculable range of a program may be greatly restricted in this case.

5.2 Analysis of Execution Efficiency

First, we consider the execution efficiency of a program obfuscated by our scheme. A single assignment instruction is replaced with assignment instructions to all the m encoded variables in Step 5 (a) of our scheme. The size of references to variables also increases by a factor of m because of Step 5 (b). The size of the obfuscated program is therefore about m^2 times that of the original one. However, the merging of assignment instructions in Step 6 (b) can reduce the number of instructions. In the best case, the number of instructions can be reduced by a factor of n, since the number of original variables is n. Ultimately, the size of an obfuscated program is at least m^2/n times that of the original one.

The number of assignment instructions and references to variables is not increased by the encoding procedure in Sato, et al.'s scheme. However, instructions are replaced with more complicated expressions. Thus, there is a loss in execution efficiency.

5.3 Experiment

We investigated experimentally the change in program size when the number of target variables n and the number of encoded variables m are altered.

The target was the source code of a program that outputs the l-th term of the Fibonacci sequence given the input l. The code was written in C, and the number of lines was 16. Five variables were contained in this code. However, only four variables could be encoded, since one variable was used for storing the input value. We wrote ten versions by applying our scheme to the code using the parameters n=1, 2, 3, 4 and m=1, 2, 3, 4, where $m \geq n$. We also wrote four versions by applying Sato's scheme with n=1, 2, 3, 4 for comparison. Finally, we obtained an executable program by compiling the original code and the obfuscated code, and investigated their functionality, efficiency, and the *complexity of variable dependences*.

Table 2 Number of instructions in the assembly code.

[Our Scheme]

$m\backslash n$	1	2	3	4
1	59	-	-	-
2	63	82	-	-
3	71	107	122	-
4	78	134	169	209

[Sato, et al.'s scheme]

$m\backslash n$	1	2	3	4
1	59	62	65	69

Table 3 *Complexity of variable dependences* in obfuscated codes.

[Our Scheme]

$m\backslash n$	1	2	3	4
1	9	-	-	-
2	12	12	-	-
3	15	21	16	-
4	18	29	24	23

[Sato, et al.'s scheme]

$m\backslash n$	1	2	3	4
1	9	9	9	9

(1) Functionality

We checked whether the correct output was returned for the inputs $l = 1, 2, 5, 10, 20, 46$ by the obfuscated codes. The original program returned the correct output for the inputs $l \leq 46$.

In our scheme, the outputs of the ten obfuscated programs for the input $n = 1, 2, 5, 10$ were identical to that of the original program. We found that the functionality of the program was not changed. However, the outputs of some obfuscated programs (where (n, m) was $(3, 4)$ or $(4, 4)$) for the input $l = 20$ differed from that of the original program, and the outputs of all the obfuscated programs given the input $l = 46$ differed from that of the original one.

With Sato, et al.'s scheme, the outputs of all the obfuscated programs for the inputs $l = 1, 2, 5, 10, 20$ were correct. However, the outputs of all the obfuscated programs for the input $l = 46$ were incorrect.

Thus, our scheme and Sato, et al.'s scheme restrict the calculable ranges.

(2) Efficiency

We compared the execution efficiency in terms of the number of instructions in the assembly code. We compiled the source code with gcc 4.01 on Fedora Core release 4 with the option '-S -O0'. The number of instructions in the original version was 57. **Table 2** shows the number of instructions in the assembly code obtained from the obfuscated codes and the original one.

With our scheme, the number of instructions increased with the number of target variables n and the number of encoded variables m. The increase was smaller than the estimate m^2/n given in Section 5.2 since some parts of the code were not affected by our obfuscation.

With Sato, et al.'s scheme, the increase was about 28 percent for all inputs l.

(3) Complexity of Variable Dependences in Obfuscated Codes

We define the *complexity of variable dependences* in the appendix. The *complexity of variables dependences* in the original code was 8. **Table 3** shows the changes in the complexity.

With our scheme, the *complexity of variable dependences* increased as the number of encoded variables m increased. Moreover, it increased as the number $m - n$ increased. That is, it became large when n was small and m was fixed. We believe this was because when $m - n$ is large, the number of non-trivial relational equations becomes large. Thus, it is difficult to specify which variable is decoded by a decoding rule.

With Sato, et al.'s scheme, the complexity of variable dependence did not increase, even if the number of target variables increased, since a variable is encoded into exactly one variable in this scheme. Thus, their scheme obfuscates only operations between variables, and does not obfuscate dependences between variables.

6. Implementation

We implemented our scheme as an obfuscation tool for C/C++ code. The tool accepts the original code of a function, and outputs the obfuscated code. The tool is written in C++, and the total size of the source code is 512 KB.

Figure 8 shows the module structure of our tool. Our tool works roughly as follows. First, the wizard GUI module is launched. The user inputs a target code. Next, the `parsing engine` analyzes the code and finds the variables that can be encoded. After that, the user inputs target variables and parameters (a matrix and a vector) for encoding the variables. The `obfuscation module` transforms the code using the parameters. Finally, the `output module` displays the original program and the obfuscated program (**Fig. 9**). The `log`

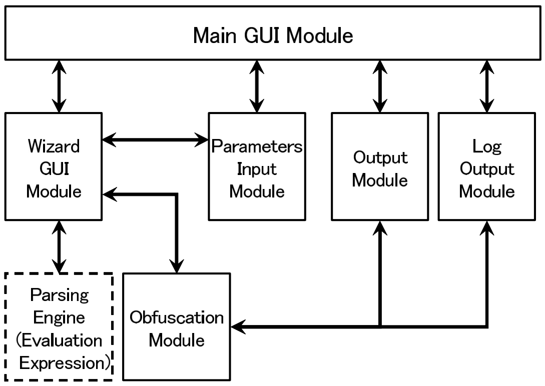
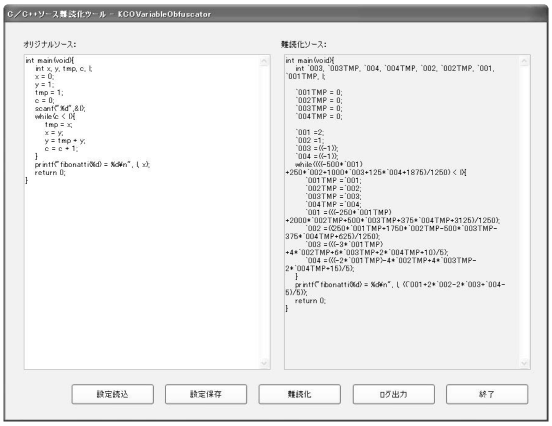


Fig. 8 Module structure of our obfuscation tool.



オリジナルソース:

```
int main(void){
    int x, y, tmp, c, l;
    x = 0;
    y = 1;
    tmp = 1;
    c = 0;
    scanf("%d",&l);
    while(c < l){
        tmp = x;
        x = y;
        y = tmp + y;
        c = c + 1;
    }
    printf("fibonatti(%d) = %d¥n", l, x);
    return 0;
}
```

難読化ソース:

```
int main(void){
    int `003, `003TMP, `004, `004TMP, `002, `002TMP, `001,
`001TMP, l;

    `001TMP = 0;
    `002TMP = 0;
    `003TMP = 0;
    `004TMP = 0;

    `001 =2;
    `002 =1;
    `003 =((-1));
    `004 =((-1));
    while((((-500*`001)
+250*`002+1000*`003+125*`004+1875)/1250) < l){
        `001TMP =`001;
        `002TMP =`002;
        `003TMP =`003;
        `004TMP =`004;
        `001 =(((-250*`001TMP)
+2000*`002TMP+500*`003TMP+375*`004TMP+3125)/1250);
        `002 =((250*`001TMP+1750*`002TMP-500*`003TMP-
375*`004TMP+625)/1250);
        `003 =(((-3*`001TMP)
+4*`002TMP+6*`003TMP+2*`004TMP+10)/5);
        `004 =(((-2*`001TMP)-4*`002TMP+4*`003TMP-
2*`004TMP+15)/5);
    }
    printf("fibonatti(%d) = %d¥n", l, ((`001+2*`002-2*`003+`004-
5)/5));
    return 0;
}
```

Fig. 9 GUI of our obfuscation tool.

`output module` outputs the log file, which contains the names of the target variables and the encoded variables, the matrix A, and the vector $\boldsymbol{b}$. Our tool can transform a 200-line code within five seconds.

7. Discussion

Here, we compare our scheme to other obfuscation schemes that transform data structures, and then we discuss the limitations of our scheme and ways in which it can be improved.

7.1 Comparison

7.1.1 Comparison with Other Proposed Schemes

Collberg, et al. describe many obfuscation schemes that can transform data structures in Section 7 of their paper[4]: *change encoding*, *promote variables*, *split variables*, *convert static to procedural data*, *merge scalar variables*, *restructure arrays*, *modify inheritance relations*, and *ordering transformation*. Sato, et al.'s scheme is an improved version of *change encoding*. We discuss whether the other schemes can protect against the three types of attack we defined in Section 2.2.

(1) Data-Dependency Attack

Promote variables, *split variables*, *restructure arrays*, *modify inheritance relations*, and *ordering transformations* can protect against this type of attack because these schemes make it more difficult to analyze the dependences between variables in a code. For example, $f(i)$ is used instead of i to determine the ith element of a list in *ordering transformation*. The attacker must additionally investigate the function f to understand the code.

(2) Dynamic Attack

None of these schemes can fully protect against this type of attack. For example, attackers can obtain the values of arbitrary variables at any point by inserting `printf` at that point.

However, obfuscation schemes that change the number of variables are somewhat effective against such attacks. In our future work, we will compare the effectiveness of these schemes to that of our scheme.

(3) Instruction-Guessing Attack

Split variables and *merge scalar variables* can protect against this type of attack because the number of variables is changed by these schemes. However, the other schemes cannot protect against this sort of attack since the number of variables is not changed. That is, with these schemes there is exactly one corresponding variable in the obfuscated code for every original variable.

(4) Numerical Attack

Split variables, *merge scalar variables*, and promote variables may not protect against this sort of attack because the original data and the obfuscated data are not numerically related in these schemes. However, the other schemes protect against this type of attack.

(5) Brute Force Attack

This type of attack cannot succeed against these schemes because they do not use secret keys.

7.1.2 Comparison with Sato, et al.'s scheme

Our scheme is an expansion of Sato, et al.'s scheme, since their scheme is a special case of our scheme where the matrix A is a diagonal matrix.

In Sato, et al.'s scheme, all non-diagonal elements of matrix A are 0. The value of an original variable affects exactly one encoded

Table 4 Instructions that must be replaced with standard instructions.

Generic name	Instructions	Replacement Instructions
Multiple-variable assignment	`a = b = 100;`	`a = 100; b = 100;`
Compound assignment	`a += 3;`	`a = a + 3;`
Pre-increment/decrement	`b = ++a;`	`a = a + 1; b = a;`
Post-increment/decrement	`b = a++;`	`b = a; a = a + 1;`
Declaration and initialization	`int a = 10;`	`int a; a = 10;`

variable. Thus, there exists exactly one encoded variable for every original variable; that is, there is a one-to-one correspondence between each original variable and each encoded variable. An attacker can use these relations through a *data-dependency attack*, *instruction-guessing attack*, or *numerical attack* as defined in Section 2.2.

In our scheme, each non-diagonal element is not necessarily 0. The data that a variable holds can be distributed among multiple encoded variables. Thus, no one-to-one correspondence between each original variable and each encoded variable exists. The advantages of our scheme arise from the following facts:

- The dependences between variables in an obfuscated code are complex.
- A single encoded variable corresponding to an original variable does not exist.
- The encoding rules for an original variable cannot be derived from only the decoding rule for the variable.

These three points respectively account for the resistance against *data-dependency attacks*, *instruction-guessing attacks*, and *numerical attacks*.

7.2 Limitations and Improvements to Our Scheme

Here, we outline some of the limitations of our scheme and our ideas for improvement.

(1) Applicability

Our scheme does not affect the control flow in a target code. Thus, we can apply our scheme to code that includes conditional branches or loops. We show specific ways to do this below.

`if` statements, `switch` statements `while` loops, and `do-while` loops have a conditional statement to determine which instruction is executed next. Target variables may be referred to in the statement. These variables and the instructions in the statement block are encoded using our scheme as described in Section 4.2.

`for` loops have three statements; the first and third ones are assignment instructions, and the second one is a conditional statement. The variables referred to in the second statement and the instructions in the loop block and the first and third statements are encoded.

Some code contains nested statements or loops. In this case, we must encode all the conditional states and instructions contained in the nested loops.

Finally, some loops contain `goto` statements, `break` statements, and `continue` statements. We do not have to deal with these statements since they do not refer to variables and do not assign values to variables.

On the other hand, our scheme cannot be applied to particular instructions. The instructions shown in **Table 4** must be replaced with standard instructions in which a variable to be updated appears on the left side and the variables that are referred to appear on the right side.

(2) Availability

Our scheme is suitable for code with many variables, but not for code with few variables.

We can apply our scheme to code with few variables, though, by introducing a variable to control the flow of the code. That is, we can encode the control variable using our scheme. Furthermore, we can introduce dummy variables into the target code and encode these variables as well.

(3) Functionality

Both Sato, et al.'s scheme and our scheme may restrict the calculable range of a program. We can use an exclusive-or operation for encoding to avoid this restriction. That is, we use exclusive-or and a Boolean matrix instead of addition and an integer matrix, respectively. The calculable range is not restricted in the improved scheme, since the exclusive-or does not cause overflows.

Each element of a Boolean matrix is restricted to a value of 0 or 1, though, so their rank tends to be small. The number of Boolean matrices which can be used for encoding is less than the number of integer matrices. Thus, the improved scheme may introduce the following problems:

- The generating algorithm for a Boolean matrix is complicated.
- The improved scheme is weaker against a

brute force attack than our original scheme, since the number of available matrices is restricted.

(4) Efficiency

The execution efficiency of our scheme is lower than that of Sato, et al.'s scheme. Thus, there is a trade-off between the execution efficiency and the *complexity of variable dependences* in an obfuscated code.

In the toy example, the number of instructions in the assembly code increased by a factor of about 3.67. However, we can improve the execution efficiency by applying our scheme only to the parts where the highest level of security is required. If we apply our scheme to ten percent of the program, the increase in the number of instructions will stay at about 37 percent.

(5) Implementation Issues

We have two implementation issues.

(5)-1 Parsing Engine

Our tool uses "expression evaluation" 5) as a parsing engine. This tool was originally used for analyzing mathematical expressions. Thus, our tool cannot deal with some C/C++ specific expressions (Table 4). We must improve the parsing engine in our future work.

(5)-2 Efficiency of Our Tool

Our tool transforms 200 lines of code within five seconds, so it can efficiently transform this amount of code. Fujiwara has stated that a function should be written in at most 100 lines 6), so according to this standard our tool is practical. In our tool, the costliest process is the derivation of the encoding variables. We will refine the algorithm in our future work.

8. Conclusion

We have proposed an obfuscation scheme that encodes variables to make the derivation of algorithms and secret data harder. Our scheme simultaneously transforms multiple variables using a matrix and a vector, and the data a variable holds is distributed among multiple variables. Thus, our scheme resists attacks better than that of Sato, et al. We have implemented our scheme as an obfuscation tool for C/C++ code. In our future work, we will remove the limitations of our scheme described in section 7.2.

References

1) Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. and Yang, K.: On the (Im) possibility of Obfuscating Programs, *Proc. Advances in Cryptology, CRYPTO 2001*, Lecture Notes in Computer Science, Vol.2139, pp.1–18 (2001).

2) Chan, J.T. and Yang, W.: Advanced obfuscation techniques for Java bytecode, *Journal of Systems and Software*, Vol.71, No.1–2, pp.1–10 (2004).

3) Collberg, C. and Thomborson, C.: Watermarking, tamperproofing, and obfuscation — Tools for software protection, *IEEE Transactions on Software Maintenance*, Vol.28, No.6 (2002).

4) Collberg, C., Thomborson, C. and Low, D.: A taxonomy of obfuscation transformations, Technical Report of Dept. of Computer Science 148, University of Auckland (1997).

5) Farkas, Z.: Expression Evaluation (1999). Available at http://www.codeguru.com/Cpp/Cpp/cpp_mfc/parsing/article.php/c843

6) Fujiwara, H.: Diagnostic room for C programming. Available at http://www.pro.or.jp/fuji/mybooks/cdiag/cdiag.10.3.html (in Japanese).

7) Gannod, G.C. and Cheng, B.H.C.: Using informal and formal techniques for reverse engineering programs with pointers, *Proc. 12th Automated Software Engineering Conference* (1997).

8) Monden, A., Takada, Y. and Torii, K.: Methods for Scrambling Programs Containing Loops, *IEICE Trans. Inf. Syst., Part 1* (*Japanese Edition*), Vol.J80-D-I, No.7, pp.644–652 (1997).

9) Ogiso, T., Sakabe, Y. and Soshi, M.: Software obfuscation on a theoretical basis and its implementation, *IEICE Transactions on Fundamentals*, No.1, pp.176–186 (2003).

10) Orlowski, A.: iTunes DRM cracked wide open for GNU/Linux Seriously, The Register NewsLetter (2004). Available at http://www.theregister.co.uk/2004/01/05/itunes_drm_cracked_wide_open/

11) Sakabe, Y., Soshi, M. and Miyaji, A.: Java obfuscation with a theoretical basis for building secure mobile agents, *Proc. Seventh IFIP TC-6 TC-11 Conference on Communications and Multimedia Security*, CMS'03, Lecture Notes in Computer Science, Vol.2828, pp.89–103 (2003).

12) Sato, H., Monden, A. and Matsumoto, K.: Program Obfuscation by Coding Data and Its Operation, *Technical Report of IEICE*, Vol.102, No.743, pp.13–18 (2003).

13) Sosonkin, M., Naumovich, G. and Memon, N.D.: Obfuscation of design intent in object-oriented applications, *Digital Rights Management Workshop*, pp.142–153 (2003).

14) Wang, C., Hill, J., Knight, J. and Davidson, J.: Software tamper resistance: Ob-

fuscating static analysis of programs, Technical report sc-2000-12, Department of Computer Science, University of Virginia (2000).

Appendix

A.1 Complexity of Variable Dependences

We present quantitatively our evaluation method which is the *complexity of variable dependences* in a code. The *complexity of variable dependences* is based on the *difficulty of variable analysis* (DVA) for each variable, which depends on the following two recursive conditions.

- The DVA of a variable is high if the variable refers to many variables.
- The DVA of a variable is high if the variable refers to variables with high DVA.

First, we show how to calculate the DVA of a variable. Then, we define the *complexity of variable dependences* in a code. The DVA of each variable is calculated as follows:

Step 1 Construct a Dependence Graph
First, nodes x_1, x_2, $\ldots$, x_n are arranged. These nodes correspond to all the variables used in a code. Edges are then added according to the dependences between variables. For example, if y is assigned a value that depends on x, we add an edge from node x to node y.

Step 2 Set Initial Values
We set initial values to the nodes. We set an initial value of 1 to nodes corresponding to variables that are initially assigned constants in the original code. We set the initial value of the other nodes to 0.

Step 3 Calculate the Node Values
The following three processes are executed until the flags are set for all nodes.

Process 1 List Nodes
First we list nodes to which a positive value (not 0) is set. Processes 2 and 3 are executed for the listed nodes.

Process 2 Add Evaluation Values
We add the value of a listed node x to the evaluation value of the adjacent nodes x_{j_1}, x_{j_2}, $\ldots$, x_{j_k}. The value of node x itself may also be updated by adding the values of other nodes. In this process, the value before updating is added to the value of adjacent nodes. Similarly, if a listed node has a self-looping edge, its own value before updating is added.

Process 3 Set Flags
We set a flag for node x_i. The flag shows the node has already been evaluated. Even if the dependence graph has a loop, checking for a flag ensures each node is evaluated only once.

Step 4 Find the DVA for each variable
The value assigned to a node is the DVA of the corresponding variable.

Finally, the *complexity of variable dependences* in the code is defined by summing the DVA values of all variables in the code.

A.2 Toy Example

We show a toy example of applying our scheme and Sato, et al.'s scheme to an algorithm; Fig. 2 shows an original code that computes the l-th term of the Fibonacci sequence, and Fig. 3 shows an original code that checks the input and determines whether to execute the function `LicenseOK()`.

A.2.1 Sato, et al.'s Scheme

We use the encoding rules $X = 2x + 2$, $Y = -y$, $\mathrm{TMP} = \mathrm{tmp} - 2$, and $C = 2c + 1$ to encode four variables in the original code (Fig. 2). The obfuscated code is shown in Fig. 1.

Furthermore, we use the encoding rules $X = 2x + 1$, $Y = y + 5$, and $K = 2k + 1$ to encode three variables in the original code (Fig. 3). The obfuscated code is shown in Fig. 4.

A.2.2 Our Scheme

We use the matrix

$$A = \begin{pmatrix} 0 & 1 & -1 & 0 \\ 2 & 0 & 1 & 1 \\ 0 & -1 & 1 & 1 \\ -2 & 1 & 1 & -1 \end{pmatrix}$$

and the vector $\boldsymbol{b} = (-2\ \ 2\ \ 1\ \ 0)^T$ to encode the variables x, y, tmp, and c in the original code (Fig. 2). The obfuscated code is shown in Fig. 6.

Furthermore, we use the matrix

$$A = \begin{pmatrix} 1 & 2 & -1 \\ -2 & 1 & 0 \\ 1 & 1 & -1 \end{pmatrix}$$

and the vector $\boldsymbol{b} = (3\ \ -1\ \ 5)^T$ to encode the variables x, y, and k in the original code (Fig. 3). The obfuscated code is shown in Fig. 7.

(Received November 29, 2005)
(Accepted June 1, 2006)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.498–512.)

Kazuhide Fukushima received his M.E. in Information Engineering from Kyushu University, Japan, in 2004. He joined KDDI and has been engaged in research on digital rights management technologies, including software obfuscation and key-management schemes. He is currently a researcher of the Information Security Lab. in KDDI R & D Laboratories Inc. He is a member of IEICE and ACM.

Shinsaku Kiyomoto received his B.E. in Engineering Sciences and his M.E. in Materials Science from Tsukuba University, Japan, in 1998 and 2000, respectively. He joined KDD (now KDDI) and has been engaged in research on stream ciphers, cryptographic protocols, and mobile security. He is currently a research engineer of the Information Security Lab. in KDDI R & D Laboratories Inc. He received his doctorate of engineering from Kyushu University in 2006. He received the Young Engineer Award from IEICE in 2004. He is a member of JPS and IEICE.

Toshiaki Tanaka received B.E. and M.E. degrees in communication engineering from Osaka University, Japan, in 1984 and 1986, respectively. He joined KDD (now KDDI) and has been engaged in research on cryptographic protocols, mobile security, digital rights management, and intrusion detection. He is currently a senior manager of the Information Security Lab. in KDDI R & D Laboratories Inc. He is a member of IEICE.