Towards Secure Cloud Computing Architecture – A Solution Based on Software Protection Mechanism –

Kazuhide Fukushima Shinsaku Kiyomoto
Yutaka Miyake
KDDI R&D Laboratories, Inc.
Fujimino, Saitama, Japan
{ka-fukushima,kiyomoto,miyake}@kddilabs.jp

Abstract

Cloud computing grows as an important keyword to accelerate IT businesses. A feature of cloud computing is distributed architecture based on unfixed nodes, and security risks have been highlighted for cloud computing services. In this paper, we present a novel approach for solving the essential issue of cloud computing: how to protect a program running in an untrusted cloud computing environment. We apply a separation technique to the program and divide it into two pieces: a user program and a protected program. Our security analysis shows that both internal and external attacks require exponential computational costs; that is, our scheme is computationally secure against these attacks.

1 Introduction

Cloud computing is an important key phrase in accelerating IT businesses. A feature of cloud computing is distributed architecture based on unfixed nodes. Cloud computing reduces the total cost of a service by sharing all computational resources with other services. Platform-as-a-Service (PaaS) is the service model of cloud computing, and it provides a program-executable environment for service providers. PaaS facilitates deployment of programs without the cost and complexity of buying and managing the underlying hardware and software layers. However, some security risks have been highlighted for cloud computing services. It is impossible for users to verify the trustworthiness of all cloud computing environments, and the concern is that operations in cloud computing may be carried out without trusted environments. The dynamic and fluid nature of the environments will make it difficult to maintain the consistency of security and ensure the ability to audit records. Thus, moving critical programs and sensitive data to a public and shared cloud computing environment is a major concern for service providers [19].

In this paper, we focus on the main issue: how to protect service programs against a malicious cloud platform provider. The malicious cloud platform provider may analyze the inside of a service program and steal important information.

We present a novel approach for solving the essential issue of cloud computing: how to protect a program running in an untrusted cloud computing environment. We apply a separation technique to the program and divide it into two pieces: a user program and a protected program. The user program executes in a trusted environment and transforms the input and output. The protected program runs in the untrusted environment and deals only with transformed data. We define two types of attacks and analyzed the computational costs of the attacks for the scheme. We evaluate the performance of our scheme using a theoretical analysis. The rest of the paper is organized as follows: section 2 provides related work about our separation approach. Section 3 explains security issues handled in the paper. Our methodologies are presented in section 4. Section 5 provides security and performance analyses. The paper concludes with section 6.

Journal of Internet Services and Information Security, volume: 1, number: 1, pp. 4-17

2 Related Work

Existing software protection schemes fall into three categories: software-based, hardware-based, and combined hardware and software based.

SOFTWARE-BASED SCHEME. License checking using an activation code is a common function of copy protection software. Software obfuscation schemes transform the original program into an obfuscated program that is difficult to analyze, while preserving its function. Some of the obfuscation schemes focus on obscuring the data structures in a program [5, 22, 10], and some focus on obscuring the control flow [12, 24, 3]. Software watermarking schemes embed auxiliary information for identification into a program [4, 17, 6]. The drawback of watermarking is that it cannot prevent or detect unauthorized copying in real time.

Online schemes use an external server to check the user license [16, 8] or to execute essential parts of a program [27, 18]. The drawbacks of these schemes are that the program manufacturer must deploy and manage a server, and program execution requires network access between the server and user terminal.

There is the same security issue on mobile agent [7, 13]: how to protect execution code against a malicious execution environment. Sander and Tschudin proposed a solution [20] using a homomorphic encryption scheme. In their scheme, all operations are executed for encrypted data; thus, only limited operations can be implemented, and heavy computations are required for the operations.

HARDWARE-BASED SCHEME. Hardware-based schemes execute a whole program on a special hardware device, such as a secure processor [11, 14, 25, 23, 21]. These schemes are expensive solutions since tamper-proof, high-performance hardware is needed to execute an entire program in a protected domain on the hardware.

COMBINED HARDWARE AND SOFTWARE SCHEME. Combined hardware and software schemes reduce the deployment cost of hardware-based schemes. A program is executed on an unprotected device and only the important functions are executed on resource-constrained hardware. Atallah and Li [2] proposed a license management scheme for smart cards. In this scheme, the program is unprotected on the device. Mana and Pimentel [15] and Zhang and Gupta [26] presented schemes where only the essential parts of the program are executed on secure hardware. Their schemes require the development of distinct functions on each piece of hardware.

Anderson [1] studied the theoretical aspects of the combined schemes based on complexity theory. A program can execute securely in conjunction with a partnering oracle on a secure device. The oracle collaborates with any program by changing the parameters, since the function of the oracle is common to all programs. Anderson proved that Turing machines are secure under the assumption that a secure device can be used.

Fukushima et al. [9] proposed a practical software protection scheme that transforms a general program into a protected program. The protected program handles encoded data on an unprotected device; then, the TPM decodes the execution result. Two open issues exist: (1) how to protect against dynamic attacks where the attacker can eavesdrop or tamper with the communication between the program and the hardware, and (2) how to apply the scheme to other variables besides integer variables. Another software protection issue is how to provide quantitative analysis of security.

3 Threats on the Cloud Computing

First, we consider adversary model for the cloud computing. Figure 1 shows typical architecture of cloud computing. In PaaS services, the platform provider supplies a software development kit (SDK) and service providers develop service programs for the platform. Users can access the services by executing these programs from a user terminal via the Internet. This paper focuses how to establish a secure

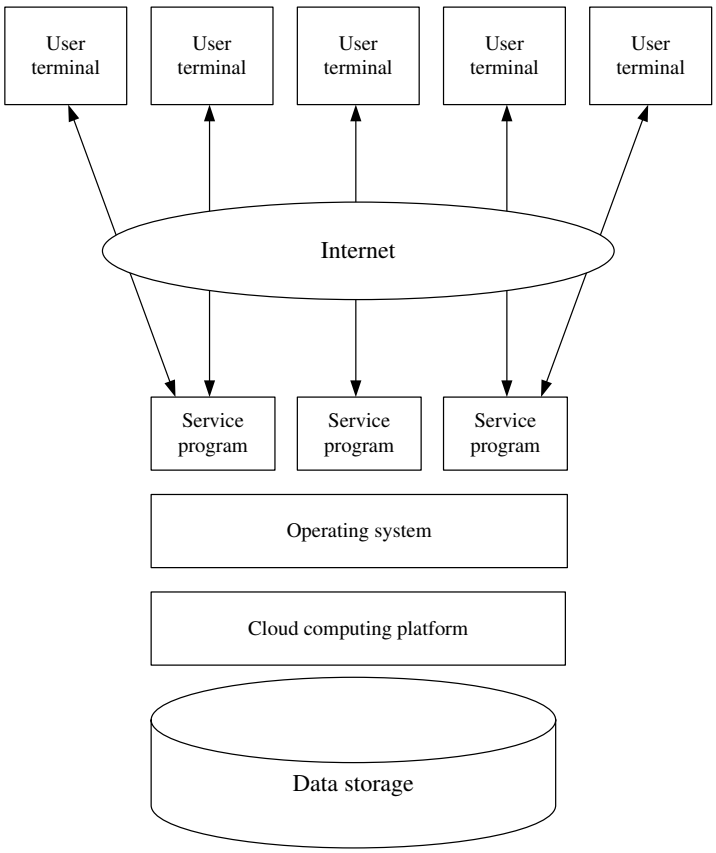

Figure 1: System Architecture

program execution platform for PaaS services.

It is assumed that three kinds of entities try to attack the cloud computing. External attackers can eavesdrop or modify Internet communications between a user terminal and service program. Malicious users try to attack other users to steal secret information or using a service without the correct permission. Furthermore, we have to consider malicious platform providers. However, if the ability of the malicious platform provider is unlimited, we have to assume all possible attacks by the provider, which is a very difficult task to realize secure cloud computing. Thus, we consider a reasonable adversary model as follows;

Curious Platform Provider. The platform provider honestly executes user requests and cannot obtain any information from the execution environment such as physical memory. However, the platform provider may try to use the user's program maliciously or to obtain information from data storage. This model is a reasonable model where we consider the system manager of the platform as an attacker.

We should consider the following threats for secure cloud computing.

- Malicious users or malicious platform providers may access a service program and execute it on the platform.
- Malicious users or malicious platform provider may steal user's information stored into the service program.
- External attacker may modify a communication between a user terminal and the platform, or steal user's information from communication data.

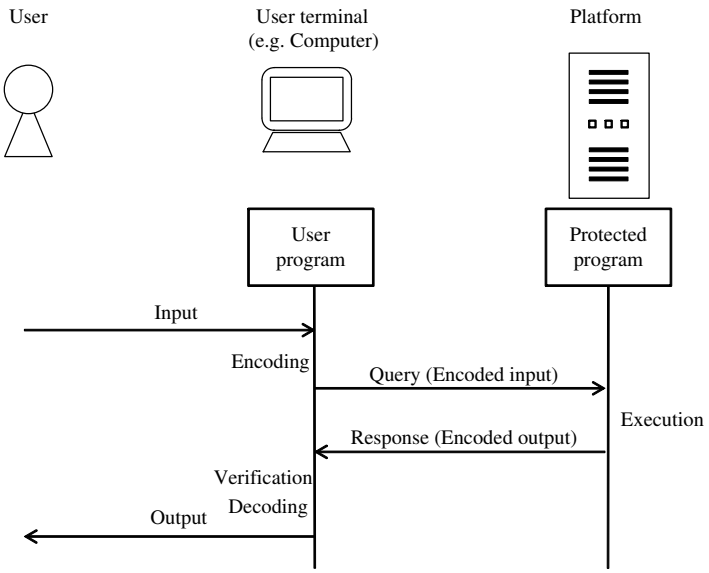

Figure 2: Overview of our scheme

4 Software Protection Scheme for Cloud Computing

Section 4.1 gives an overview of our scheme. A program transformation procedure and its extension are shown in Section 4.2 and Section 4.3. We explain the execution sequence in Section 4.4.

4.1 Overview

Our scheme transforms a target program into a protected program and a user program. The protected program is executed on the platform and only handles encoded data. The program receives encoded input and sends back the encoded output to the user program. The user program is executed on the user terminal. This function encodes the input by the user and sends it to the protected program. After receiving the encoded output, the user program checks the validity of the output. If it is valid, the function returns the decoded execution result to the user. Figure 2 shows the execution process of the whole program. The user program encodes the input using encoding rules and checks the validity of the data received from the protected program using a non-trivial relation. Finally, it decodes the execution result of the whole program using a decoding rule. The encoding rules, non-trivial relation, and decoding rule are defined in the program transform procedure in Section 4.2.

Throughout this paper, the following assumptions are used:

- The target program returns the execution result as a numerical value.
- No attack against the user program on the user terminal is possible.

4.2 Program Transformation Procedure

The procedure encodes multiple variables with the same bit-lengths l. The limitation on the length of variables is eliminated by the extension in Section 4.3. We denote a left bit-rotation operator by $\lll$ and a bitwise exclusive-or operator by $\oplus$.

Step 1. Select Target Variables We select n variables whose values are to be encoded. Note that the variable that stores the execution result must be selected — we call the variable x_1. The variables storing user input also must be selected. Let these target variables be $x_1, x_2, \ldots, x_n$. Then, we select rotation amounts $s_1, s_2, \ldots, s_n$ $(0 \leq s_i < l)$ for each target variable. The target variable x_i is bit-rotated by s_i bits in accordance with the encoding rules.

Step 2. Define Encoded Variables We define m encoded variables $y_1, y_2, \ldots, y_m$ to save the encoded values of the target variables. The number of encoded variables m must be greater than the target variables n. We then select rotation amounts $t_1, t_2, \ldots, t_m$ $(0 \leq t_j < l)$ for each encoded variable. The encoded variable y_j is bit-rotated by t_j bits in accordance with the encoding rules.

Step 3. Construct Encoding/Decoding Rules and Non-trivial Relations We select an $m \times n$ Boolean matrix A with rank n and an m-dimensional vector b. Each element of the matrix must be 0 or 1, and each element of the vector must be an l-bit constant. Next, make a relational equation using matrix A, vector b, and rotation amounts $s_1, s_2, \ldots, s_n, t_1, t_2, \ldots, t_m$ as follows:

$$\begin{pmatrix} y_1 \lll t_1 \\ y_2 \lll t_2 \\ \vdots \\ y_m \lll t_m \end{pmatrix} = A \begin{pmatrix} x_1 \lll s_1 \\ x_2 \lll s_2 \\ \vdots \\ x_n \lll s_n \end{pmatrix} \oplus b.$$

The encoding rules $(E_1, E_2, \ldots, E_m)$ are obtained by solving this equation for encoded variables y_1, y_2, $\ldots$, y_m:

$$\begin{aligned} y_1 &= E_1(x_1, x_2, \ldots, x_n), \\ y_2 &= E_2(x_1, x_2, \ldots, x_n), \\ &\vdots \\ y_m &= E_m(x_1, x_2, \ldots, x_n). \end{aligned}$$

The user program on the user terminal stores all the encoding rules to encode the input given by a user.

Then, we solve the above equation for the target variables $x_1, x_2, \ldots, x_n$. We have decoding rules $(D_1, D_2, \ldots, D_n)$ such that

$$\begin{aligned} x_1 &= D_1(y_1, y_2, \ldots, y_m), \\ x_2 &= D_2(y_1, y_2, \ldots, y_m), \\ &\vdots \\ x_n &= D_n(y_1, y_2, \ldots, y_m). \end{aligned}$$

n out of m encoded rules are used to find the decoding rules. The user program stores decoding rule D_1 to decode the execution result stored in variable x_1.

Non-trivial relations $(R_1, R_2, \ldots, R_{m-n})$ such that

$$\begin{aligned} R_1(y_1, y_2, \ldots, y_m) &= 0, \\ R_2(y_1, y_2, \ldots, y_m) &= 0, \\ &\vdots \\ R_{m-n}(y_1, y_2, \ldots, y_m) &= 0, \end{aligned}$$

are obtained by replacing target variables x_i with $D_i(y_1, y_2, \ldots, y_m)$ in the unused $m-n$ rules. Encoded variables $y_1, y_2, \ldots, y_m$ satisfy these relational equations. Thus, we can use a relational equation to check the validity of the output from the protected program in the platform. Recall that a query consists of the values of the encoded variables. Using these non-trivial relations, a decoding rule can be expressed in 2^{m-n} ways. For example, $D_1 \oplus R_1$ or $D_1 \oplus R_2 \oplus R_3$ is another expression for decoding rule D_1. Note that we define exclusive-or of two functions $F_1 \oplus F_2(y_1, y_2, \ldots, y_m)$ by $F_1(y_1, y_2, \ldots, y_m) \oplus F_2(y_1, y_2, \ldots, y_m)$.

Step 4. Encode Target Variables in Program This step substitutes all the variables $x_1, x_2, \ldots, x_n$ used in the program with the encoded variables $y_1, y_2, \ldots, y_m$. We replace the target variables with the assigned values, and then we replace the variables whose values are referenced. Note that variable x_1 in the instruction that returns the execution result is not replaced in this step.

First, we encode the variables to be assigned. We replace the assignment instructions for variables $x_1, x_2, \ldots, x_n$ with assignment instructions for the encoded variables $y_1, y_2, \ldots, y_m$. Generally, the assignment instruction $x_i \leftarrow v$ is replaced with the following instruction:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \leftarrow \begin{pmatrix} E_1(x_1, x_2, \ldots, v, \ldots, x_n) \\ E_2(x_1, x_2, \ldots, v, \ldots, x_n) \\ \vdots \\ E_m(x_1, x_2, \ldots, v, \ldots, x_n) \end{pmatrix}.$$

Encoding rules in which variable x_i is replaced with v appear on the right side of the instruction. Encoded variables $y_1, y_2, \ldots, y_m$ are simultaneously updated when this instruction is executed.

Next, we encode the variables to be referenced. We replace target variables $x_1, x_2, \ldots, x_n$ with the decoding rules $D_1, D_2, \ldots, D_n$, respectively. The instruction obtained in the previous step is changed to the following instruction:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} \leftarrow \begin{pmatrix} E_1(D_1, D_2, \ldots, v, \ldots, D_n) \\ E_2(D_1, D_2, \ldots, v, \ldots, D_n) \\ \vdots \\ E_m(D_1, D_2, \ldots, v, \ldots, D_n) \end{pmatrix}.$$

Step 5. Modify Program We modify the protected program so that it executes in conjunction with the user program in the user terminal. This step consists of three parts: (1) adding declarations, (2) replacing assignment instructions, and (3) adding communication instructions.

First, we add the declaration of encoded variables $y_1, y_2, \ldots, y_m$, and temporary variables $z_1, z_2, \ldots, z_m$. Temporary variables are used to save the values of the encoded variables.

Next, we replace the simultaneous assignment instructions introduced in Step 4 with instructions for one variable that can be implemented in a typical program. For example, the above assignment instruction is decomposed into the following instructions:

$$\begin{aligned} z_1 &\leftarrow y_1, z_2 \leftarrow y_2, \ldots, z_m \leftarrow y_m, \\ y_1 &\leftarrow E_1(D'_1, D'_2, \ldots, v, \ldots, D'_n), \\ y_2 &\leftarrow E_2(D'_1, D'_2, \ldots, v, \ldots, D'_n), \\ &\vdots \\ y_m &\leftarrow E_m(D'_1, D'_2, \ldots, v, \ldots, D'_n). \end{aligned}$$

The values of variables $y_1, y_2, \ldots, y_n$ are saved to temporary variables $z_1, z_2, \ldots, z_m$; then, the updated value of each variable is sequentially calculated. $D'_1, D'_2, \ldots, D'_n$ denote the decoding rules where the encoded variable y_j is replaced with temporary variable z_j.

Finally, we insert the instruction for collaboration between the protected program and user terminal. The instruction sends values $v_1, v_2, \ldots, v_m$ of encoded variables $y_1, y_2, \ldots, y_m$ to the user program of the user terminal. This instruction is inserted into the end of the protected program and the instruction that outputs the value of x_1 as the execution result.

4.3 Extension for Arbitrary Length Variables

The bit lengths of all the variables were assumed to be l in the program transfer procedure. However, actual programs contain character, integer, and double data types. Generally, each data type has a distinct data length; thus, the program transformation procedure cannot be applied without some modification. This subsection addresses how to encode variables longer or shorter than l bits.

We divide long data into multiple l-bit data. For example, a union data type is available in the C/C++ language. We can assign nl-bit data to n l-bit variables and encode these variables according to the program transformation procedure. A l-bit variable can store shorter data with no loss of bits. Thus, we can assign the shorter data to an l-bit variable and encode the variables.

4.4 Execution

We show the execution sequence of the whole program. The user program on the user terminal stores all the encoding rules E_1, E_2, ..., E_m decoding rule D_1, and non-trivial relation R_1. The non-trivial relation checks the validity of the response from the protected program, and the decoding rule decodes the execution result. (There is no special reason to select R_1 for checking. We can use other non-trivial relations $R_2, R_3, \ldots, R_{m-n}$ instead of R_1.)

1. The user provides input to the user program on the user terminal.
2. The user program set the input to the corresponding target variables. Random values are set to the remaining target variables as dummy input.
3. The user program encodes the input.
4. The user program sends the encoded input as a query to the protected program on the platform.
5. The protected program processes the encoded input.
6. The protected program returns the encoded output.
7. The protected program sends back the encoded output as a response to the user program.
8. The user program checks the validity of the response in order to eliminate modified responses from an attacker. The user program calculates the value of $R_1(v_1, v_2, \ldots, v_m)$ from the response. If $R_1(v_1, v_2, \ldots, v_m) = 0$ holds, the user program decides the response is valid. Otherwise, it decides that the query is modified.
9. The user program decodes the value of x_1 if the response from the protected program is valid. The user program uses decoding rule D_1, i.e., $x_1 = D_1(v_1, v_2, \ldots, v_m)$.
10. The user program returns the value of x_1 to the user. If the response from the protected program is invalid it returns $\perp$.

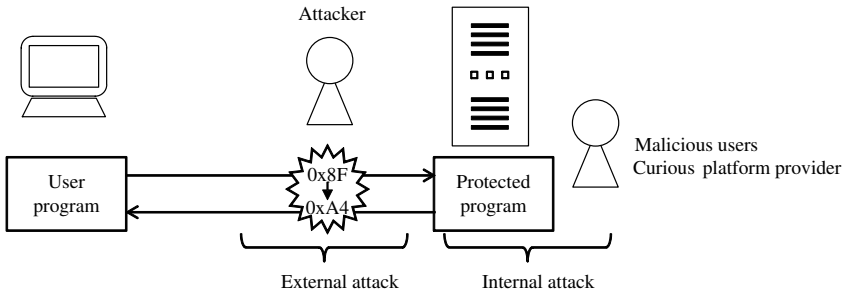

Figure 3: Our Attack Model

5 Analysis

We show the analyses of security and the performance for our scheme in this section.

5.1 Security Analysis

We quantitatively evaluate the computational cost of attacks against the protected program.

5.1.1 Attack Model and Security Definition

We use the following attack models for the security analysis. There are two attack models: an internal attack and an external attack. An internal attack can be executed by as malicious platform provider and malicious users who can access the protected program on the platform. An external attack can be executed by an outside attacker who can only access the communication between the user terminal and the platform. Figure 3 shows our attack model. The security definition is provided as follows:
Definition. *A cloud computing architecture is secure if the architecture can prevent an internal attack and an external attack.*

5.1.2 Security against Internal Attack

The goal of an attacker is to use the protected program or obtain user information stored in the program. An internal attack can be categorized as a brute force attack and an algebraic attack. A brute force attack and an algebraic attack try to derive the encoding rules. After obtaining all the encoding rules, the attacker can use the protected program by transforming its input and output or decode the user information in the program. We do not consider dynamic attacks since a curious platform provider cannot access the execution environment (see our definition in Section 3).

In a brute force attack, the attacker must guess all the elements of matrix A and vector b; and rotation amounts $s_1, s_2, \ldots, s_n, t_1, t_2, \ldots, t_m$. The number of possible $m \times n$ matrices with rank n is greater than 2^{mn-2} [10], and the number of possible vector b is 2^l. The number of possible combinations of rotations is l^{m+n}. Thus, the computational cost of a brute force attack is $2^{mn-2} l^{m+n} 2^l = \Omega(\exp(mn))$.

In an algebraic attack, an attacker collects m decoding rules and solves the equations for the target variables $x_1, x_2, \ldots, x_n$. The attacker must guess the correspondence between m decoding rules and n target variables. Recall that there are 2^{m-n} expressions for each decoding rule, thus distinct rules may decode the same target variables. The guess requires $n^m = \Omega(\exp(m \log n))$ computational cost. If an attacker derives all the non-trivial relations, the number of equations required for the analysis is reduced to n. Non-trivial relations need to be identified by a brute force attack. The attacker must guess the presence or absence of each encoded variable y_j; then, the attacker must guess the rotation

amount t_j if y_j exists. The attacker also guesses the l-bit constant in the relation. Thus, the guessing of non-trivial relations requires $(l+1)^m \cdot 2^l / l = \Omega(\exp(m \log l))$ computational cost. Note that there are l valid expressions for a non-trivial relation. We have another expression for a non-trivial relation by bit-rotating both sides of the relation by i bits $(1 \leq i \leq l-1)$. Finally, the total computational cost of this attack is $\Omega(\exp(m \log l))n^n = \Omega(\exp(m \log l + n \log n))$. We conclude that internal attacks require $\Omega(\exp(\min\{m \log n, m \log l + n \log n\}))$ computational cost, considering the costs for the attacks discussed above.

5.1.3 Security against External Attack

The goal of an attacker is to decrypt or tamper the communication between a user terminal and the platform without accessing the protected program. An external attack can be categorized as a decrypting attack and a tampering attack.

In a decrypting attack, an attacker tries to obtain user input or output by decrypting communications. The attacker cannot access the protected program in the platform, and one has to derive all the encoding rules by brute force. Thus, the computational cost of a decrypting attack is $2^{mn-2} l^{m+n} 2^l = \Omega(\exp(mn))$ from the above analysis.

In a tampering attack, an attacker has to derive the decoding rule D_1 in order to bypass the verification by the user program. The total number of possible candidates for decoding rule D_1 is $(l+1)^m \cdot 2^l$ considering the presence of each variable, rotation amounts, and constants. There are 2^{m-n} valid expressions for decoding rule D_1. A tampering attack requires $(l+1)^m \cdot 2^l / 2^{m-n} = \Omega(\exp(m(\log l - 1) + n))$ computational cost.

We conclude that external attacks require $\Omega(\exp(m(\log l - 1) + n))$ computational cost. Furthermore, an external attacker cannot determine the identity of encoded input. Note that random numbers are set to target variables (see section 4.4).

5.2 Performance Analysis

The number of instructions increases in Step 4 of the program transformation procedure. The first procedure replaces variables to be assigned with encoded variables. An assignment instruction is replaced with m instructions for each encoded variable. This instruction also contains at most m bit-rotation operations and exclusive-or operations. The number of instructions increases with $O(m^2)$.

The second procedure of Step 4 replaces variables to be referenced with encoded variables. A target variable is replaced with the corresponding decoding rule. A decoding rule contains at most m bit-rotation operations and exclusive-or operations; thus, the number of instructions increases with $O(m)$. The protected program needs $O(m^3)$ times instructions as the original program. However, we can prevent the increase in the number of instructions by simultaneously encoding independent instructions.

A protected program contains many bit-rotation operations, but the C/C++ language does not provide bit-rotation operations as a standard function. In contrast, some processors support bit-rotation operations as processor instructions. In such cases, we may reduce the program execution time by using this instruction. In-line-assembly techniques are available to embed processor instructions into the source code.

The user program encodes the input and decodes the output of the protected program. Thus, two round-trip communications between a user terminal and the platform are required for every execution. The user program has to store all the encoding rules $E_1, E_2, \ldots, E_m$, decoding rule D_1 and non-trivial relation R_1. The total number of possible decoding rules or non-trivial relations is $(l+1)^m \cdot 2^l$; that is, $\lg((l+1)^m \cdot 2^l)$ bits of data are needed to identify D_1 or R_1. Thus, the user program consume at most $(m+2)[m(\lg l + 1) + l]$-bits of storage of a user terminal. $E_1, E_2, \ldots, E_m$, D_1 and R_1 contain at most m

bit-rotation operations and m exclusive-or operations. Thus, our scheme imposes $O(m^2)$ computational cost on the user terminal.

6 Conclusion

In this paper, we presented a possible solution for security issues in cloud computing environments. Our scheme divides an execution code into two pieces and protects the whole program against malicious environments. We quantitatively analyzed the security of our scheme. Our results show that our scheme achieves computational security against both internal and external attacks.

References

[1] W. Erik Anderson. On the secure obfuscation of deterministic finite automata. In *Cryptology ePrint Archive, 2008/148*, 2008.

[2] Mikhail J. Atallah and Jiangtao Li. Enhanced smart-card based license management. In *Proc. of IEEE International Conference on E-Commerce (CEC'03), Newport Beach, California, USA*, pages 111–119. IEEE, June 2003.

[3] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. of the 4th Information Security Conference (ISC'01), Malaga, Spain, LNCS*, volume 2200, pages 144–155. Springer-Verlag, October 2001.

[4] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), San Antonio, Texas, USA*, pages 311–324. ACM Press, January 1999.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Computer Science, University of Auckland, 1997.

[6] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation — tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[7] A. Corradi, R. Montanari, and C. Stefanelli. Security issues in mobile agent technology. In *Proc. of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '99), Cape Town, South Africa*, pages 3–8. IEEE, December 1999.

[8] Ori Dvir, Maurice Herlihy, and Nir N. Shavit. Virtual leashing: Internet-based software piracy protection. In *Proc. of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, USA*, pages 283–292. IEEE, June 2005.

[9] Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. Obfuscation mechanism in conjunction with tamper-proof module. In *Proc. of the 12th IEEE International Conference on Computational Science and Engineering (CSE'09), Vancouver, Canada, vol. 2*, pages 665–670. IEEE, August 2009.

[10] Kazuhide Fukushima, Shinsaku Kiyomoto, Toshiaki Tanaka, and Kouichi Sakurai. Analysis of program obfuscation schemes with variable encoding technique. *IEICE Transactions on Fundamentals*, E91-A(1):316–329, 2008.

[11] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater. An architecture of security management unit for safe hosting of multiple agents. In *Proc. of the International Workshop on Intelligent Communications and Multimedia Terminals, Ljubljana, Slovenia*, pages 79–82, November 1998.

[12] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Proc. of Mobile Agents and Security 1998, LNCS*, volume 1419, pages 92–113. Springer-Verlag, 1998.

[13] Yang Kun, Guo Xin, and Liu Dayou. Security in mobile agent system: problems and approaches. *SIGOPS Oper. Syst. Rev.*, 34:21–28, January 2000.

[14] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectual support for copy and tamper resistant software. In *Proc. of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00), Cambridge, Massachusetts, USA*, pages 168–177. ACM Press, November 2000.

[15] Antonio Mana and Ernesto Pimentel. An efficient software protection scheme. In *Proc. of 16th IFIP International Conference on Information Security (ISC'01), Paris, France*, pages 385–401. Kluwer Academic Publishers, June 2001.

[16] Microsoft Corporation. Technical overview of windows rights management services for windows server 2003.

[17] Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *Proc. of the 24th Computer Software and Applications Conference (COMPSAC'00), Taipei, Taiwan*, pages 191–197. IEEE, October 2000.

[18] Saadia Mumtaz, Sameen Iqbal, and Engr. Irfan Hameed. Development of a methodology for piracy protection of software installations. In *Proc. of the 9th IEEE International Multi-topic Conference (INMIC'05), Karachi, Pakistan*, pages 1–7. IEEE, December 2005.

[19] Kresimir Popovic and Zeljko Hocenski. Cloud computing security issues and challenges. In *Proc. of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO'10), Opatija, Croatia*, pages 344 –349, May 2010.

[20] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Proc. of Mobile Agents and Security 1998, LNCS*, volume 1419, pages 44–60. Springer-Verlag, 1998.

[21] Weidong Shi, Hsien-Hsin, S. Lee Laura Falk, and Mrinmoy Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proc. of the 33rd International Symposium on Computer Architecture (ISCA'06), Boston, Massachusetts, USA*. IEEE, June 2006.

[22] A.V. Shokurov. An approach to quantitative analysis of resistance of equivalent transformations of algebraic circuits. Technical report, Institute for System Programming Russian Academy of Sciences, 2004.

[23] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proc. of the 17th annual international conference on supercomputing (ICS'03), San Francisco, California, USA*, pages 160–171. ACM Press, June 2003.

[24] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Proc. of the31st IEEE/IFIP International Conference of Dependable Systems and Networks (DSN'01), Göteborg, Sweden*, pages 193–202. IEEE, June-July 2001.

[25] Jun Yang, Jun Yang Youtao, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), San Diego, California, USA*, pages 351–360. IEEE, December 2003.

[26] Xiangyu Zhang and Rajiv Gupta. Hiding program slices for software security. In *Proc. of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO'03), Chamonix, France*, pages 325–336. IEEE, April 2003.

[27] Jianming Zhao, Nianmin Yao, and Shaobin Cai. A new method to protect software from cracking. In *Proc. of World Congress on Computer Science and Information Engineering (CSIE'09), Los Angeles, California, USA*, pages 636–638. IEEE, March-April 2009.

Kazuhide Fukushima received his M.E. in 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 at the Information Security Laboratory of KDDI R&D Laboratories Inc. He received his doctorate in engineering from Kyushu University in 2009. He is a member of ACM, IEICE, and IPSJ.

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 senior researcher at the Information Security Laboratory of KDDI R&D Laboratories Inc. He was a visiting researcher of the Information Security Group, Royal Holloway University of London from 2008 to 2009. He received his doctorate in engineering from Kyushu University in 2006. He received the IEICE Young Engineer Award in 2004. He is a member of JPS and IEICE.

Yutaka Miyake received the B.E. and M.E. degrees in electrical engineering from Keio University, Japan, in 1988 and 1990, respectively. He joined KDD (now KDDI) in 1990 and has been engaged in the research on high-speed communication protocols and secure communication system. He received his doctorate in engineering from the University of Electro-Communications, Japan, in 2009. He is currently a senior manager at the Information Security Laboratory in KDDI R&D Laboratories Inc. He received the IPSJ Convention Award in 1995 and the Meritorious Award on Radio of ARIB in 2003.

A Toy Example

This section shows a toy example. Figure 4 shows the source code of the original program that calculates the i-th term of the Fibonacci sequence. Figure 5 shows the source code of user program and Fig. 6 shows the source code of the protected program.

We use these encoding rules

$$\begin{aligned} y_1 &= b \lll 28 \oplus i \lll 4 \oplus -315671005, \\ y_2 &= i \lll 14 \oplus a \lll 22 \oplus 1966373211, \\ y_3 &= a \lll 3 \oplus b \lll 19 \oplus -1857746742, \\ y_4 &= a \lll 28 \oplus b \lll 12 \oplus i \lll 20 \oplus 824205888 \end{aligned}$$

to encode variables a, b and i in the target program. Decoding rules are as follows:

$$\begin{aligned} a &= y_1 \lll 20 \oplus y_4 \lll 4 \oplus -315671005, \\ b &= y_2 \lll 26 \oplus y_4 \lll 20 \oplus -1751883256, \\ i &= y_3 \lll 5 \oplus y_4 \lll 12 \oplus -983868959. \end{aligned}$$

The encoded variables y_1, y_2, y_3 and y_4 satisfy the non-trivial relation

$$R_1(y_1, y_2, y_3, y_4) = y_1 \lll 9 \oplus y_2 \lll 31 \oplus y_3 \lll 12 \oplus -942915997 = 0.$$

```
int main(void){
  int a, b, cnt, tmp, i;
  a = 0;
  b = 1;
  printf("Input i:  ");
  scanf("%d", &i);
  for(cnt = i; cnt > 0; cnt--){
    tmp = a;
    a = b;
    b = tmp + b;
  }
  printf("Fibonacci(%d) = %d\", i, a);

  return 0;
}
```

Figure 4: Source code of target program

```
int UserProgram(void){
  int a, b, i, y1, y2, y3, y4;
  int ary[4];

  printf("Input i:  ");
  scanf("%d", &i);

  a = rand();  b = rand();
  y1 = rotl(b,28)^rotl(i, 4)^-315671005;
  y2 = rotl(i,14)^rotl(a,22)^1966373211;
  y3 = rotl(a, 3)^rotl(b,19)^-1857746742;
  y4 = rotl(a,28)^rotl(b,12)^rotl(i,20)^188669860;

  ary[0] = y1; ary[1] = y2; ary[2] = y3; ary[3] = y4;
  ProtectedProgram(ary);
  y1 = ary[0]; y2 = ary[1]; y3 = ary[2]; y4 = ary[3];

  if ((rotl(y1, 9)^rotl(y2,31)^rotl(y3,18)^-942915997) == 0){
    printf("Fibonacci(%d) = %d\n", i, rotl(y1,20)^rotl(y4, 4)^1372793011);
  }
  else {
    printf("Verification Error !\n");
  }

  return 0;
}
```

Figure 5: Source code of user program

```
int ProtectedProgram(int *ary){
  int y1, y2, y3, y4, z1, z2, z3, z4, cnt, tmp;

  y1 = ary[0]; y2 = ary[1]; y3 = ary[2]; y4 = ary[3];

  // a = 0 ;
  // b = 1 ;
  z1 = y1; z2 = y2; z3 = y3; z4 = y4;
  y1 = rotl(z3, 9)^rotl(z4,16)^-1466261441;
  y2 = rotl(z3,19)^rotl(z4,26)^-1555298291;
  y3 = -1857222454;
  y4 = rotl(z3, 25)^z4^-1792894447;

  // for (cnt = i; cnt > 0; cnt--){
  for (cnt = rotl(y3, 5)^rotl(y4,12)^-983868959; cnt > 0; cnt--){

    // tmp = a;
    tmp = rotl(y2,10)^rotl(y3,29)^rotl(y4, 4)^807417485;

    // a = b;
    z1 = y1; z2 = y2; z3 = y3; z4 = y4;
    y1 = rotl(z2,22)^rotl(z3, 9)^837019167;
    y2 = rotl(z3,19)^rotl(z4,26)^rotl(z2,16)^rotl(z4,10)^560590101;
    y3 = rotl(z2,29)^rotl(z4,23)^rotl(z2,13)^rotl(z4,7)^1034069037;
    y4 = rotl(z2,22)^rotl(z4,16)^rotl(z2,6)^rotl(z3,25)^1518028104;

    // b = tmp + b;
    z1 = y1; z2 = y2; z3 = y3; z4 = y4;
    y1 = rotl(tmp + (rotl(z2,26)^rotl(z4,20)^-1751883256),28)^rotl(z3, 9)
          ^rotl(z4,16)^-1197825985;
    y2 = rotl(z3,19)^rotl(z4,26)^rotl(z1,10)^rotl(z4,26)^-1885831993;
    y3 = rotl(z1,23)^rotl(z4, 7)^rotl(tmp + (rotl(z2,26)^rotl(z4,20)
          ^-1751883256),19)^534531408;
    y4 = rotl(z1,16)^rotl(tmp + (rotl(z2,26)^rotl(z4,20)^-1751883256),12)
          ^rotl(z3,25)^-1606436710;
  }

  ary[0] = y1; ary[1] = y2; ary[2] = y3; ary[3] = y4;

  return 0;
}
```

Figure 6: Source code of protected program