Software Protection via Hiding Function using Software Obfuscation

Venus Samawi[1] and Adeeb Sulaiman[2]
[1]Department of Computer Science, Al al-Bayt University, Jordan
[2]College of Administrative Science, Applied Science University, Kingdome of Bahrain

Abstract: *Application Service Provider (ASP) is a business that makes computer-based services (small and medium sized businesses) available to clients over a network. The usual ASP sells a large application to large enterprises, but also, provides a pay-as-you-go model for smaller clients. One of the main problems with ASP is the insufficient security to resist attacks and guarantee pay-as-you-go. Function hiding can be used to achieve protection for algorithms and assure charging clients on per-usage basis. Encryption functions that can be executed without prior decryption (function hiding protocol) gives good solution to the problems of software protection. Function hiding protocol faces a problem if the same encryption scheme is used for encrypting some data about the function and also, the output of the encrypted function. In such case, an attacker could reveal the encrypted data easily thereby comprising its confidentiality. This paper aims to develop a software protection system based on function hiding protocol with software obfuscation that overcomes function hiding protocol problems. The suggested system is a multi-client system that allows charging clients on a per-usage basis (pay-as-you-go) and satisfies both confidentiality and integrity for the ASP and the client.*

Keywords: *Software protection, function hiding, software obfuscation, ASP.*

Received July 10, 2011; accepted May 22, 2012; published online August 5, 2012

1. Introduction

Application Service Providers (ASPs) have evolved from the increasing costs of dedicated software of small to medium sized businesses. With ASPs, the costs of such software can be lowered. At the same time, the problem of upgrading has been reduced from the client by placing the services-upgrade responsibility on the ASP. There are several forms of ASP businesses. For instance, functional ASP distributes a single application, such as credit card payment processing or time-sheet services. An enterprise ASP delivers broad spectrum solutions. A local ASP delivers small business services which provide a pay-as-you-go mode. To provide an ASP offering, the vendor must also, provide a secure product [18]. One of the approaches that could be used to assure charging clients on per-usage basis and provide certain level of security is through the usage of a function hiding protocol. The key point of function hiding is to encrypt a special class of functions such that they remain executable and produce encrypted result to prevent clients from copying and using the program without paying anything for it.

In a function hiding protocol, the client executes the protected program with encrypted coefficients. The client will not get the clear-text results until he sends the encrypted results to the producer (who charges the client) to decrypt them and sends clear-text result back to the client. The encryption technique used is probabilistic Goldwassr and Micali [11, 15] with two supporting algorithms Plus and Mixed-mult that are used to allow encrypted function to be executed without requiring prior decryption [16].

Function hiding protocol needs to guarantee the secrecy of its coefficients, especially when the same key is used for encrypting the coefficients of the function and the output of the encrypted function. Such situation allows the attacker to reveal the encrypted coefficients easily. This problem is called coefficient attack problem. Instead of sending outputs of the program to the producer, the client (attacker) sends the encrypted coefficients that he finds in the program. The client may even scramble them by multiplication with some random quadratic residue, such that producer cannot recognize these values as the hidden function coefficients (polynomial coefficients). According to the function hiding protocol, the producer has to decrypt the encrypted data (in attacking case, the sent data is the encrypted polynomial coefficients) and thus would supply the client the main information (original coefficients values), which must be kept secret. Coefficient attack problem is general problems that function hiding schemes have to solve.

In this paper, we tackle the problem of coefficient attacking (mentioned above) by:

1. Using obfuscation technique in this research, the resistance to the reverse engineering process is enhanced by adding session termination property in case of time expiration, and/or rule violation.
2. Making use of hash-table and Greatest Common

Devisor (GCD) to assure that the decrypted data does not contains >70% of the polynomial coefficients.

To provide security to the clear-text results generated by the producer before transmitting them to the client, authentication process is provided. To prove the authenticity of the service provider (producer), the clear text results are encrypted using public key (its private key known only to the client), then encrypted with private key of the producer.

Furthermore, a detailed description of the implementation of the function hiding process is given. Nine algorithms are written to build the developed protection system in addition to the used obfuscation technique. This system is tested with three different applications and proved secure. The tests are carried on stations of a LAN. We comprehensively survey, analyze ASP security and pay-as-you-go problems and how hiding function within software could provide certain level of software security.

The rest of this paper is arranged as follows: section 2 concerned with how function hiding aid the software protection system. The aspects of the function hiding design are discussed. Some key approaches and techniques that are useful in the construction of function hiding in addition to the necessary mathematical concepts are presented in detail. The developed software protection system is illustrated in section 3. The realistic threat model, which indicates what a cracker is able to do, is discussed in section 4. Section 5 discusses software obfuscation, its importance and techniques. Evaluation and testing of the developed software protection system are presented in section 6. Section 7 illustrates how multi-clients are handled in the suggested system. Finally, we conclude in section 8.

2. Software Protection via Function Hiding

Main applications for code privacy are found in the software industry and with service providers that seek for methods to make copying or learning proprietary algorithms technically impossible. For instance, for ASP and mobile software agents (designed to be executed on different hosts with different environmental security conditions). It is important to provide protection against various attacks such as unauthorized access to private data, malicious modification of its code etc. Function hiding can be used to accomplish software protection against disclosure and ensures that only licensed users are able to acquire the clear-text output of the protected software [12, 18]. The basic steps of function hiding protocol are illustrated in Figure 1 [15].

Let *E* be a mechanism to encrypt a function *f* implemented in a program *P* where Alice (producer) and Bob is (client):

1. The producer encrypts *f*, and creates a program *P(E (f))*
2. Producer sends software *P(E(f))* to the client.
3. Client executes *P(E(f))* with input x and sends the result *(E[R])* back to the producer
4. Producer decrypts *(E[R])*, obtains *R* and sends the result *(R)* back to the client.

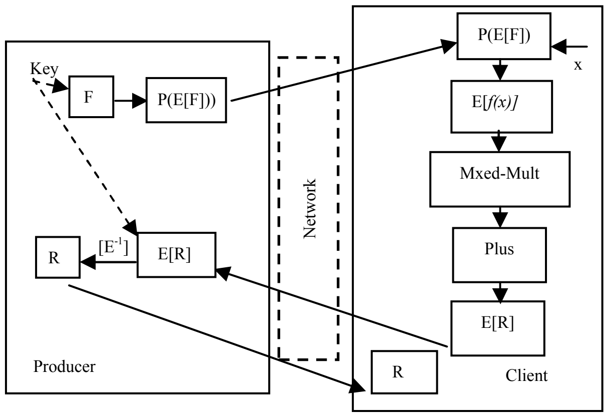

Figure 1. A basic protocol for executing encrypted functions [19].

Based on the above protocol, software producer can charge clients on a per-usage basis. To implement such a technique, additive homomorphism scheme could be used to enable hiding of a polynomial function in a program. Before illustrating the suggested model of software protection, the public-key and probabilistic public-key are discussed. Since function hiding protocol is based on Goldwasser-Micali scheme, it is important to illustrate some needed mathematical principles.

2.1. Public Key and Probabilistic Public-Key Systems

Public-Key crypto system is introduced by Diffie and Hellman in 1976. In such system, user *A* has a public encryption transformation E_A with a public key (P_A) saved in a public key directory to be used by others to encrypt messages before sends them to *A*; and a private decryption transformation D_A used to decipher the received messages, known only to user *A*, secrecy and authenticity are provided by separate transformations.

The public key crypto systems RSA and Knapsack schemes are deterministic in the sense that under a fixed public key, a certain plain text *m* is always has some or one of the following [4, 14]:

1. The scheme is not secure for all probability distributions of the message space.
2. It is sometimes easy to compute partial information about the plaintext *m* from the cipher text *c*.
3. It is easy to detect when the message sent twice.

Public-key encryption scheme is said to be polynomial secure if no passive adversary can, in expected polynomial time, select two plaintext messages *m1* and *m2* with probability significantly >0.5 [4, 11, 14].

Public key encryption scheme is said to be significantly secure if, for all probability distributions over the message space, whatever a passive adversary can compute in expected polynomial time about the plaintext given the cipher text, it can also, compute in expected polynomial time without the cipher text [4, 11, 14].

The probabilistic public-key encryption [11, 14] has some differences from the public key cryptosystems, these are, the encryption decryption operations are performed on binary numbers, quadratic residue principle and Jacobi symbols are used to get the public key, and does not produce the same encrypted result when repeating the encryption operation more than once, so it is none deterministic.

2.2. Mathematical Background

In this section, mathematical principles needed in the implementation of the proposed system are illustrated. These include quadratic residue, rings, relatively prime numbers, Jacobi symbol, additively homomorphic encryption, and polynomial rings.

- *Quadratic Residue [14]:* Let $a \in Z^*_n$, a is said to be a quadratic residue modulo n, or a square modulo n, if there exists an $x \in Z^*_n$ such that $x^2 \equiv a (\bmod n)$. If no such x exists, then a is called a quadratic non-residue modulo n. The set of all quadratic residues modulo n is denoted by Q_n, and the set of all quadratic non-residues is denoted by $\overline{Q_n}$.
- *Rings[10]:* A ring $<R, +, .>$ is a set R together with two operations + and ., which is called addition and multiplication respectively, defined on R such that the following axioms are satisfied:

  R_1: $<R, +>$ is an Abelian group,
  R_2: multiplication is associative,
  R_3: for all $a, b, c \in R$,

  left distribution law: $a(b+c)=(ab)+(ac)$, and right distributive law: $(a+b)c=(ac)+(bc)$, holds.
- *Relatively Prime Numbers [14]:* Two integers a and b are said to be relatively prime or coprime if GCD (a, b)=1, where GCD is the greatest common divisor.
- *Legendre Symbol and Jacobi Symbol [14]:* The Legendre symbol is a useful tool for keeping track of whether or not an integer a is a quadratic residue modulo a prime number p:

Let p be an odd prime and a is an integer.The Legendre symbol $\left(\frac{a}{p}\right)$ is defined for $a \geq 0$ and p odd prime where:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \mid a \\ 1 & \text{if } a \in \underline{Q_p} \\ -1 & \text{if } a \in Q_p \end{cases} \qquad (1)$$

- *Jacobi Symbol [14]:* Let $n \geq 3$ be odd with prime factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. The Jacobi symbol $\left(\frac{a}{n}\right)$ is defined to be:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \cdots \left(\frac{a}{p_k}\right)^{e_k} \qquad (2)$$

  Observe that if n is prime number, then the Jacobi symbol is just the Legendre symbol.
- *Additively Homomorphic Encryption [13, 15]:* Let R and S be ring function $E{:}R \rightarrow S$ is called additively homomorphic if there is an efficient algorithm Plus to compute $E(x+y)$ from $E(x)$ and $E(y)$ that does not reveal x and y.
- *Polynomial Rings [1]:* If R is a commutative ring, then a polynomial in the indeterminate x over the ring R is an expression in the form:

$$f(x) = a_0 + a_1x^1 + a_2x^2 + a_3x^3 + \ldots + a_nx^n \qquad (3)$$

  where each $a_i \in R$ and $n \geq 0$. The element a_i is called the coefficient of x_i in $f(x)$. The largest integer m for which $a_m \neq 0$ is called the leading coefficient of $f(x)$. If $f(x)=a_0$ (a constant polynomial) and $a_0 \neq 0$, then $f(x)$ has degree 0. If all the coefficients of $f(x)$ are 0, then $f(x)$ is called the zero polynomial and its degree, for mathematical convenience, is defined to be $-\infty$. The polynomial f(x) is said to be monic if its leading coefficient is equal to one. Each polynomial is composed of a number of monomials. A monomial in x is an expression of the form: ax^n. Where a and x are integer numbers. The number a is called the coefficient of the monomial. If $a \neq 0$, the degree of the monomial is n.

3. The Developed Function Hiding System

Using function hiding protocol for software protection can be defeated by coefficient attack (the elements send to the producer is in fact the encrypted coefficients). In this case, the producer will decrypt the polynomials coefficient and handed them to the client (attacker). Sander and Tschudin [15] suggested to solve this problem by making sure that the producer is able to detect if an element send to the producer was in fact produced as an output of the encrypted program (E[R]). The key idea is to hide additional polynomials (besides the function f) which simultaneously executed when P is run. The additional polynomials serve as checksums used by producer. By careful construction, it is unfeasible for a software pirate to construct numbers that pass the producer's checksum test for elements that are not outputs of the producer encrypted program. But this solution suffers from the problem of the need for additional polynomials and checksum test which takes additional time. In addition, the checksums should be easy to evaluate for producer. In particular, they should be much faster to evaluate than the original polynomial f itself.

We developed the system model shown in Figure 1 to overcome the coefficient attack problem and prove authenticity. The suggested system based on software obfuscation. The details of our system are illustrated in Figure 2.

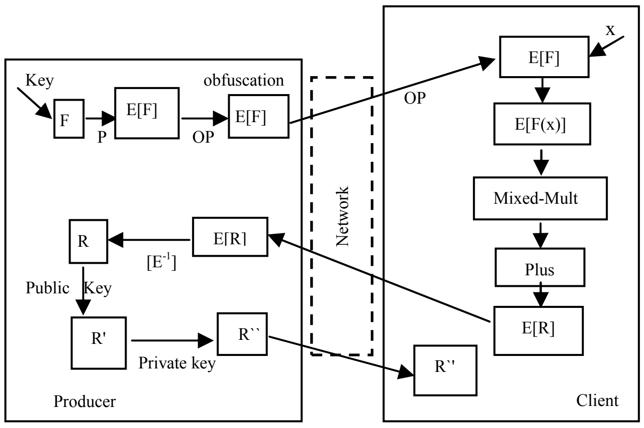

Figure 2. The proposed protocol for executing encrypted functions.

In Figure 2, *E* is the encryption function, *F* is the function to be protected, E^{-1} is the decryption function, and *R* is the result. The two functions Mixed-Mult, and Plus are the functions that are used to support the operation of function hiding. Let *E* be a mechanism to encrypt a function *f* implemented in a program *P*:

1. The producer encrypts *f* and creates a program *P(E(f))*.
2. Producer performs obfuscation on program *P* and produce Obfuscated Program (OP) (to complicate reverse software engineering process that could be used to reveal the hidden polynomial coefficients).
3. Producer sends software *OP(E(f))* to the client.
4. Client executes *OP(E(f))* at the input *x*, then use mixed multiplicative (Mixed-Mult) and an additive (Plus) encryption function to hide polynomials in a program
5. Client sends the encrypted result *(E[R])* to the producer.
6. Producer decrypts *(E[R])*, obtaining *R*.
7. To provide security for client results, encrypt *R* with public key of the client and produce *R`*.
8. To prove authenticity of the producer, encrypt *R`* by private key of producer and generate *R``*, then sends the result back to the client.

Next, let us develop the steps illustrated above. The main steps that are used to construct the function hiding system are illustrated in Algorithm 1 shown below. Other functions are called within this algorithm in order to accomplish the function hiding process which will be illustrated in the subsequent sections.

```
Algorithm 1: Function Hiding Model
Let F: be the polynomial illustrated in equation 3.
In order to hide this polynomial, the following steps are performed:
Step 1: Encrypt each coefficient ($a_1, a_2, a_3, ..., a_n$) using
        Algorithm 6 (Goldwasser-Micali probabilistic
        public-key encryption) to get $E(a_1), E(a_2), ..., E(a_n)$,
        where each element $E(a_i)$ represents a set of
        numbers resulting from encrypting each binary
        digit of the coefficient $a_i$.
Step 2: Compute $x^1, x^2, x^3, ..., x^n$.
Step 3: Compute the result of each monomial i.e. $E(a_n)\ x^n$
        using algorithm 8 (Mixed-Mult) and store the
        results in an array M; where each monomial is
        stored in a single cell of M.
Step4: Add-Up the elements of array M (Algorithm 9).
```

3.1. Encryption- Decryption Modules

Step 1 in Algorithm 1 encrypts the coefficient of the polynomial *F*. In this section, we describe the algorithms that implement in total the Goldwasser-Micali encryption method.

```
Algorithm 2: $Z^*_n$ calculation
Input:  n; such that n is an integer.
Output: Set of integers such that integer $a \in [0,..., n-1]$
where GCD(a,n)=1.
Step 1: Specify $Z_n=[0,..., n-1]$
Step 2: For each $a \in Z_n$, Do
        If GCD(a,n)=, then add a to the set of $Z^*_n$
```

```
Algorithm 3: Jacobi and legendre symbol computations
JACOBI (a, n)
Input: An odd integer $n \geq 3$, and an integer a, $0 \leq a \geq n$.
Output: The Jacobi symbol $\left(\frac{a}{n}\right)$ (and hence the Legendre
symbol when n is prime)
Step 1: If a=0 then return (0).
Step 2: If a=1 then return (1)
Step 3: Write $a=2^e a_1$, where $a_1$ is odd.
Step 4: If e is even then set s $\leftarrow$ 1.
Otherwise
set s $\leftarrow$1 if n≡1 or 7(mod 8),
set s $\leftarrow$-1 if n≡3 or 5(mod 8)
Step 5: If n≡3(mod 4) and $a_1$≡3(mod 4) then set s $\leftarrow$-s.
Step 6: Set $n_1 \leftarrow$ n mod $a_1$
Step 7: If $a_1$ = 1 then return (s);
Otherwise return (s×JACOBI($n_1$, $a_1$))
```

```
Algorithm 4: Quadratic residue modulo n test
Input: n, an integer
Output: Set of Quadratic residue Module n numbers.
Step 1: Find $Z^*_n$ using Algorithm 3.
Step 2: For each $a \in Z_n$ do;
Step 3: If $(x^2-a)$ mod n=0 $\rightarrow$ add a to the quadratic residues
        modulo n set; where x is any other integer such that
        $a \in Z_n$.
```

```
Algorithm 5: Key generation for Goldwasser-Micali
Probabilistic public key encryption
Step 1: Select two large prime numbers p and q randomly,
        where they should be roughly the same size (number
        of digits)
Step 2: Compute n=pq
Step 3: Select an integer $y \in Z_n$ such that y is a quadratic
        non-residue modulo n and the Jacobi symbol
        $\left(\frac{y}{n}\right)=1$, usingalgorithms 3 and 4.
```

```
Step 4: The public key of user A is $(n,y)$; and the privet key is
        the pair $(p,q)$.
```

Algorithm 6: Goldwassr-Micali Probabilistic Public-Key Encryption

This algorithm encrypts an integer m into t-tuple, where t is the number of binary digits of the integer m.

User A encrypts an integer m for user B, and then B will decrypt this integer.

A should perform the following steps

```
Step 1:Obtain B's authentic public key $(n,y)$, using
        algorithm 5.
Step 2: Represent the message $m$ as binary string $m=m_1, m_2,
        ..., m_t$ of length $t$.
Step 3: For $i=1$ to $t$ Do
        a. Evaluate $Z^*_n$ using algorithm 2
        b. Pick an $x \in Z_n$ at random
        c. If $m_i=1$ then set $c_i \leftarrow yx^2 \bmod n$;
    Otherwise set $c_i \leftarrow x^2 \bmod n$
Step 4: Send t-tuple $c=(c_1, c_2, ..., c_t)$ to B.
```

Algorithm 7: Goldwasser-Micali Probabilistic Public-Key Decryption

This algorithm takes t-tuple and transforms it back to an integer m; where m is the clear text. To recover the plaintext message m (of length t bits) from c, user A should do the following:

```
Step 1: For $i=1$ to $t$ Do
        a. Find the Legendre symbol $e_i=\left(\frac{ci}{p}\right)$ (algorithm 3).
        b. If $e_i=1$ then set $m_i \leftarrow 0$; otherwise set $m_i \leftarrow 1$.
Step 2: The decrypted message is $m=m_1, m_2, ..., m_t$.
```

Algorithm 8: Mixed-mult computation

Input: integer variable x (having b binary digits, such that $x=x_1...x_b$) and encryption of coefficients a; E(a).

Output: list (M) of encrypted integers.

```
Step 1: For $i = 1$ to $b$ D
        a. If $xi=1$, then compute $E(a2^i)$, using algorithm 2
        b. Put the result in list M
Step 2: Add-up elements of list M using the plus algorithm
(Algorithm 9).
```

Algorithm 9: Plus computation.

This algorithm adds up the monomials of the encrypted polynomial: $\sum_{i=1}^{n} p_i$

where each P_i is a list (M) obtained by algorithm 8.

```
Step 1: Pick a random number $x$ from $Z^*_n$, let $c = x^2 \bmod n$.
Step 2: For $j=1$ to $b$, Do steps 3-5; where $b$ is the number of
        binary digits of each number $a$.
Step 3: $Sum[j]=P_1[j].\ P_2[j] \bmod n$.
Step 4: $Sum[j]=Sum[j].\ c \bmod n$.
Step 5: If $P_1[j]$ and $P_2[j] \neq x^2 \bmod n$, then $c= y.x^2 \bmod n$.
Step 6: For $i=3$ to $m$, Do steps 7; where $m$ is the number of
        monomials in the polynomial.
Step 7: For $j=1$ to $b$, Do steps 8-10.
Step 8: $Sum[j]=Sum[j].\ P_i[j] \bmod n$.
Step 9: $Sum[j]=Sum[j]\ c \bmod n$.
Step 10: if $Sum[j]$ and $P_i[j] \neq x^2 \bmod n$,
         then $c=y.\ x^2 \bmod n$.
```

4. The Realistic Threat Model

When a security mechanism is required to achieve a security goal, it is important to illustrate the realistic threat model, which points up what a cracker is able to do. Crackers knowledge and resources could be discriminated based on [20, 21]:

- *Algorithm understanding level of the used protection mechanism:* The cracker knows the cipher algorithm, but not the secret information such as the secret key.
- *Level of system observation skill:* The cracker owns *a* binary file, disassembled code, decompiled code of *P*, as well as a computer system *M* in which *P* is executed. The cracker has a debugger with breakpoint functionality that can watch internal states of *M*, e.g., memory snapshot of *M*, audio-visual outputs of *M* and the input and output value of *P*. The cracker also, monitors the execution trace of *P* (history of executed opcodes).
- *System control skill level:* When program *P* is executed on computer system *M*, the cracker controls the mouse and keyboard inputs of *M* and run *P* with an arbitrary input values. The cracker can change the instructions and the operand values in *P*, in addition to the memory image of *M*, before and/or during running *P* on *M*.

In this work, the expected threat model is based on reverse engineering (level of system observation skill) specifically once a cracker has the binary program (executable program), he can understand the principles of the used algorithm. Also, assume that the cracker has a static analyzer such as a dis-assembler and a de-compiler, as well as a debugger (dynamic analyzer). In other words, the expected cracker has both algorithm understanding and observation skills that allow him to extract the encrypted coefficients of the hidden function.

In order to hide secrets in an implemented software and hinder reverse engineering process, a number of obfuscation techniques have been proposed based on the expected threat model [9, 21] as will discussed in the next section.

5. Software Obfuscation

Software obfuscation has become a vital mean to hide secret information that exists in software systems. Obfuscations transform a program *P* to obfuscation program OP as shown in Figure 2. OP is functionally equivalent to the original program but it is more complex and difficult to be understood [9, 21]. The most popular obfuscation techniques [7, 8, 21]:

- *Lexical obfuscations:* (e.g., comment removal, identifier renaming and debugging info removal, etc.,).

- *Data obfuscations:* Data obfuscations thoroughly change the data structure of a program and encrypt literals including modifying inheritance relations, restructuring arrays, etc. They make the obfuscated codes so, complicated, which makes it is very difficult to recreate the original source code.
- *Control-flow obfuscation:* Obfuscates the layout and control flow of binary code. Many obfuscation techniques use opaque predicates to forged infeasible control flow, and then insert fake code that obfuscates the control and data flow.

To overcome the expected threat model (illustrated in the previous section), two obfuscation techniques are used: lexical obfuscator, and changing data type obfuscator for a chosen variables. The chosen variables are the encrypted hidden function coefficients. The data type will be changed from long-term to short-term to make the data obfuscation complicated. The used approach is as follows:

1. Parse the source program (un-obfuscated program) to remove comments and find all tokens of the program.
2. Find and keep all program variables through analyzing the tokens, perform variable renaming, then
3. Choose the variables that are important to obfuscate. To obfuscate variables, choose splitting, or extending method and convert them into array of short term variables [6, 7, 8]. In this work, variable splitting is used since the obfuscation metrics (potency and resilience) of variable splitting all grow with the number of variables into which the original variable is split [3].

The resulting program is the OP. For further security, white-box cryptography [9] could also, be used.

6. System Evaluation

The proposed protocol making use of function hiding protocol based on Goldwasser-Micali scheme. Hiding a polynomial f in a program P according to the method described by Sander and Tschudin [15] exhibit secured against known cipher text attack as "P guarantees that no information is leaked about the coefficients of the polynomial f" [15]. On the other hand, there is coefficient attack problem which is (in this work) handled by obfuscating program P and generates OP. But does the OP highly resists reverse engineering process (i.e., prevent specifying the coefficients) and solves the coefficient attack problem.

As well known, secure obfuscation algorithms have been proven to be impossible [5]. Program obfuscation does not prevent software engineering attack, it will only decelerate it. So, it is a matter of time before attacker could recognize the coefficients of the polynomial. But how could we evaluate the used obfuscation scheme?

To assess the reverse engineering complexity of obfuscated code, most researchers use potency and resilience metrics. Potency is the amount of obscurity added to the code, i.e., strength of OP against a human de-obfuscator. Resilience measures strength of OP against automated de-obfuscator [5]. Others works use different approach and assess obfuscation technique through controlled experiments involving human subjects [2, 5] as will be used in this work.

- *Experimental Planning*

In this work, the attacker has complete control over the execution platform (e.g., the Java Virtual-Machine, system calls). This implies that the attacker can trace and profile the execution of OP, and can run a debugger on OP. We choose 10 high ability subjects who have experience in reverse engineering. Four experiments were carried out according to the following procedure: Each subject receives OP and data file and asked to specify the polynomial coefficients of each task. For each of the four tasks to be performed, mark the start time; write the answer; mark the stop time.

Table 1. Evolution results of 40 experiments.

	#Coefficients	**Correctness**	**Time Needed (Hours)**
Poly1	5	90%	6
Poly2	10	70%	9
Poly3	15	58%	10
Poly4	20	47%	12

The tested hypotheses related to differences in time (max time given was 12 hours/ experiment) needed to perform the tasks, and the accuracy of the task result. Table 1 shows the average results of the 40 experiment from curacy and estimated time needed to get the results. From the experiment results, the time needed to perform the tasks significantly increases and the accuracy decreases when number of coefficients increased. Upon the results, to grantee preventing the client from revealing the polynomial coefficients we decided to terminate the session in two cases:

- *The Time Stamp (Expiration Time):* The client is requested to send the data he wants to decrypt within less than 12 hours after he made the request for the service.
- *Rule-Violation:* The polynomial coefficients are saved in a hash table. After decrypting the received data, check the decrypted data with the hash table content. If it contains more than 80% of the polynomial coefficients, the session will be terminated without sending the decrypted data to the client. Using hash table needs *O(1)* as time complexity. The hash table size will depends on number of polynomial coefficients.

- To overcome the problem of scrambling the previously sent coefficients by multiplying them with some random quadratic residue so, that the server cannot recognize them as previously sent coefficient, GCDs of the received coefficients and the recently stored in the hash table are calculated. If all the results 1's, this means that the received coefficients are not multiple of the original coefficients. Otherwise, indicates a multiplication has been done. The session also, terminates without sending decrypted data to the client. The time complexity of Euclid's GCD algorithm of two integers u, v, where $u>v$ is of $O(log_2|v|)$.

7. Multi-Client System

The suggested approach is used to serve one client. To make the system able to serve many clients, the coefficients of the same service can be encrypted with different encryption functions (different modulus for each user) and coefficients obfuscated (split or merged) in different way. To prevent the same client from trying to reveal the coefficients by different sessions, for each request (session), the client will receive different copy of the application. This will prevent him from making use of multiple sessions to perform reverse engineering and overcome the Timestamp restriction.

When a client makes a request, the application is split into two sites (parts), part1 (at server site) that register client, specify Time-stamp, built a hash table for the used coefficients. Encrypt the hidden function with new modules and client special encryption key K. Obfuscate the application Program (OP). Finally, sends OP to the client (part 2). The client will run the application program and gets the encrypted result. The encrypted result will be sent to the server. The server (part 1) will check the Time-stamp, if it is expired then end the session with the client. In case no time expiration, the server will decrypt the data sent by the client, check them with the coefficient stored in the hash table. If 70% of the coefficients match, then the client request will be refused and session will be terminated. Else, part-1 will ask for the fee of the application. When money is received, the decrypted results will be authenticated (as explained before) and sent to the client, then terminate the session.

8. Conclusions

Software piracy is a major financial problem for ASPs where small enterprises can sell software on a per-usage basis. This paper is concerned with the security of ASP. We suggest a multi-client approach that makes use of the function hiding technique to achieve protection of algorithms against revelation. To prevent the same client from trying to reveal the coefficients by different sessions, the coefficients of the same service are encrypted using different encryption functions (different modulus for each user). Coefficients obfuscated (split) in different way.

The suggested approach guarantees charging clients on a per-usage basis. Moreover, we describe a protocol that ensures, under certain conditions, that only licensed users are able to gain the clear-text output of the program, thereby providing confidentiality and integrity for both ASP and client.

The proposed approach is applied to a special class of functions for which secure and computationally feasible solutions are to be obtained. The key point of this work is to encrypt functions such that they remain executable. We further improve the confidentiality of the system by making reverse engineering a difficult task. This was accomplished by: 1). using both lexical obfuscation and changing data type obfuscation method to hide any confidential data in a program, 2). Terminate session with client in case of time expiration or rule violation. The testing of the suggested approach is encouraging and it meets the intended objective. As future work, improve obfuscations using obfuscation method suggested by Wei and Ohzeki [19], and evaluation of the proposed framework with other programs.

References

[1] Auvil D., *Algebra for College Students*, McGraw-Hill, USA, 1996.

[2] Badger L., Kilpatrick D., Matt B., Reisse A., and Vleck T., "Self-Protecting Mobile Agents Obfuscation Techniques Evaluation Report," *Technical Report*, NAI Labs, 2002.

[3] Balakrishnan A. and Schulze C., "Code Obfuscation Literature Survey," available at: http://pages.cs.wisc.edu/~arinib/writeup.pdf, last visited 2005.

[4] Buchmann A., *Introduction to Cryptography*, Springer, Johannes, 2004.

[5] Ceccato M., Penta M., Nagra J., Falcarin P., Ricca F., Torchiano M., and Tonella P., "Towards Experimental Evaluation of Code Obfuscation Techniques," *in Proceedings of the 4th ACM Workshop on Quality of Protection*, USA, pp. 39-46, 2008.

[6] Chen H. and Hou T., "Changing Data Type Method of Data Obfuscation on Java Software," *in Proceedings of International Computer Symposium*, Taiwan, pp. 439-442, 2004.

[7] Chen H., Yuan L., Xi W., Zang B., Huang B., and Yew P., "Control Flow Obfuscation with Information Flow Tracking," *in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Micro-Architecture*, USA, pp. 391-400, 2009.

[8] Cho S., Chang H., and Cho Y., "Implementation of an Obfuscation Tool for C/C++ Source Code Protection on the XScale Architecture," *in Proceedings of Software Technologies for Embedded and Ubiquitous Systems*, Berlin, vol. 5287, pp. 406-416, 2008.

[9] Chow S., Eisen P., Johnson H., and Oorschot P., "A White-Box DES Implementation for DRM Applications," *in Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management*, Berlin, vol. 2696, pp. 1-15, 2002.

[10] Farleigh J., *A First Course in Abstract Algebra*, Addison-Wesley, USA, 2002.

[11] Goldwasser S. and Micali S., "Probabilistic Encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270-299, 1984.

[12] Hacini S., Guessoum Z., and Boufaïda Z., "Using a Trust-Based Environment Key for Mobile Agent Code Protection," *in Proceedings of World Academy of Science, Engineering and Technology*, pp. 854-859, 2008.

[13] Melchor A., Gaborit P., and Herranz J., "Additively Homomorphic Encryption with T-Operand Multiplications," *in Proceedings of the International Association for Cryptologic Research*, pp. 138-154, 2008.

[14] Menezes A., Oorchot P., and Vanstone S., *Handbook of Applied Cryptography*, CRC Press, USA, 1996.

[15] Sander T. and Tschudin C., "On Software Protection via Function Hiding," *in Proceedings of the 2nd International Workshop IH'98 Portland Oregon*, USA, vol. 1525, pp. 111-123, 1998.

[16] Sander T. and Tschudin C., "Toward Mobile Cryptography," *in Proceedings of Security & Privacy*, California, pp. 215-224, 1998.

[17] Seroul R., *Programming for Mathematicians*, Springer, Paris, 2000.

[18] Smith B., Campbell L., Cheah J., Lachmann A., Milstein S., Morgan D., Nartovich A., and Roelofs J., *Application Service Provider Business Model: Implementation on the iSeries Server*, International Business Machines Corporation, US, 2001.

[19] Wei Y., and Ohzeki K., "Obfuscation Methods with Controlled Calculation Amounts and Table Function," *in Proceedings of the International Multi-Conference on Computer Science and Information Technology*, Wisla, vol. 5, pp. 775-780, 2010.

[20] Yamauchi H., Kanzaki Y., Monden A., Nakamura M., and Matsumoto K., "Software Obfuscation From Crackers' View Point," *in Proceedings of the International Conference, Advances In Computer Science and Technology*, Mexico, pp. 1-6, 2006.

[21] Yamauchi H., Monden A., Nakamura M., Tamada H., Kanzaki Y., and Matsumoto K., "A Goal-Oriented Approach to Software Obfuscation," *International Journal of Computer Science and Network Security*, vol. 8, no. 9, pp. 59-71, 2008.

Venus Samawi is an associative professor in Al al-Bayt University, at the Department of Computer Science. She received her BSc from University of Technology at 1987, the MSc and PhD degrees from Computer Science Department in Al-Nahrain University (Saddam University previously) at 1992 and 1999 respectively. She supervises three PhD students in system programming, pattern recognition, and network security. She also, leads and teaches modules at both BSc and MSc Levels in computer science. She is a reviewer in four IEEE conferences (ICIEA 2009, 2011, 2012, and ICFCN'12). Her special area of research is AI, neural networks, genetic algorithms, and image processing.

Adeeb Sulaiman is an associate professor at the College of Administrative Science, Applied Science University. He holds a PhD degree in computer science University of Newcastle Upon Tyne, UK, in 1984, MSc degree in computer science, University of Glasgow, UK, 1981, and BSc degree in electrical and electronic engineering, University of Technology, Iraq in 1977. He published 12 papers in the fields of cryptography, information hiding, digital watermarking, and algorithms design. Worked at Universities in different Arab countries (Iraq, Jordan, Oman, Sudan and Bahrain). He was a head of the Departments of Computer Science, Information Systems, and Computer Communication, and as faculty member. Now, he is an assistant dean of College of Administrative Science, Applied Science University, Kingdome of Bahrain.