Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 NRF ERC

ROSAEC MEMO
2014-001
February 11, 2014

Encrypted Execution

Daejun Park, Jeehoon Kang, Kihong Heo, Sungkeun Cho,
Yongho Yoon, and Kwangkeun Yi
Seoul National University
{djpark,jhkang,khheo,skcho,yhyoon,kwang}@ropas.snu.ac.kr

February 11, 2014

Abstract

We present secret execution in which an encrypted program is evaluated without decryption, to give an encrypted result whose decryption yields the original result.

1 Introduction

Can we execute an encrypted program without decryption? In the cloud computing era, more people want to execute their programs in cloud server. The biggest challenge for delegating program execution is security—how to keep the programs confidential? One possible solution is protecting the programs via obfuscation (e.g., virtual machine-based obfuscation) [4, 15, 3, 14]. Unfortunately, however, this protection is not perfect; one can decode the obfuscation—it's only a matter of time. Fortunately, however, cryptologist has already researched similar problem for decades and recently proposed a solution: homomorphic encryption—basis of our work.

Homomorphic encryption is an encryption scheme that preserves certain operations on encrypted data. A homomorphic encryption $\mathcal{E}$ is said to preserve an operation op if it provides $\underline{op}$, an encrypted version of op, such that for a plain text m,[1]

$$\underline{op}(\mathcal{E}(m)) \equiv \mathcal{E}(op(m))$$

For example, using a homomorphic encryption that preserves addition operation, we can get the sum of encrypted data without decrypting each data, just by summing up all given encrypted data and decrypting the sum.

Recently, a powerful homomorphic encryption scheme is proposed by Gentry[7, 8, 22], in the sense that the encryption scheme preserves addition and multiplication operations, which leads to preserve arbitrary operations since we can construct arbitrary circuit with only addition and multiplication operations. This is because addition (modulo 2) and multiplication operations correspond to XOR and AND gates respectively, and all circuits can be constructed by using only XOR and AND gates. For this reason, Gentry's homomorphic encryption scheme is called a *fully* homomorphic encryption scheme.

Based on homomorphic encryption, we present secret execution in which an encrypted program is evaluated without decryption. We propose a protocol how to encrypt a given program and how to execute the encrypted program without decryption. Our secret execution protocol guarantees that 1) encrypted programs are totally secure, in the sense that attackers can never reconstruct any piece of original programs from encrypted programs, and 2) execution results are correct, whose decryption yields the very results of execution of original programs.

[1]Throughout this section, all equations are using notations defined at Section 2.1.

Contributions

- We present a cryptographic protocol for program execution. We project cryptographic concept of homomorphic encryption to our domain, programs and their execution. Although Gentry's fully homomorphic encryption scheme itself is sufficient for all operations (including program execution), our work is meaningful in that we present a specific instance for program execution, that is, we, for the first time, definitize the blurred region unexplored so far.
- More specifically, we present how to evaluate expressions with encrypted operators, how to handle memory operation (such as assignment and lookup operations) under encryption, and how to execute loop under encrypted loop condition.

1.1 Overview

A tricky problem comes from managing control flow. In conventional execution of a program, program executor find out how program control (e.g., program counter) flows. For example, at an if statement, executor need to know which part (among then-part and else-part) to be executed, at a loop, executor need to know how many times loop to be executed, at a function call, which function to be called, and so on. However, in execution of encrypted programs, we should prevent the executor from perceiving which control flow to be taken. Otherwise, control flow information is revealed, which give rise to security vulnerability. For example, as for the following *if* statement,

$$\textit{if}\ (e)\ \textit{stmt}_t\ \textit{stmt}_f$$

suppose that executor can determine which branch, $\textit{stmt}_t$ or $\textit{stmt}_f$, to be taken, say $\textit{stmt}_t$. Then, although he has no idea of the original program, executor can realize, at least, e is evaluated to be true. Accumulated throughout entire program, these revealed informations are used in statistical attack, eventually causing encryption to be broken. To sum, we make the executor to evaluate control flow without noticing which one is taken.

This seemingly ironic requirement/problem can be reduced to well-known problem, Oblivious Transfer (OT)[20, 6]. In OT, a client requests i in encrypted form, and a server returns ith data, s_i, to the client in encrypted form whose decryption yields s_i. More specifically, let us consider binary OT: a server has secret data s: s_0 and s_1 (integer value), a client requests i, either 0 or 1, in the form of encrypted one, and the server returns encrypted value of s_i to the client. We can establish a protocol to achieve the above requirements as the follows:

- Suppose an public key homomorphic encryption scheme $\mathcal{E}$ preserving addition (modulo 2) and multiplication, and also having semantic security (refer to Section 2.2 for more details) property. (e.g., Gentry's FHE[22])
- A client generates a request q according to i, either 0 or 1, such that[2]

$$q = \begin{cases} (\mathcal{E}(1), \mathcal{E}(0)) & \textit{if}\ i = 0 \\ (\mathcal{E}(0), \mathcal{E}(1)) & \textit{if}\ i = 1 \end{cases}$$

- Given a request $q = (q_0, q_1)$, either $(\mathcal{E}(1), \mathcal{E}(0))$ or $(\mathcal{E}(0), \mathcal{E}(1))$, a server generates an answer a according to q such that

$$a = q_0 \times \mathcal{E}(s_0) + q_1 \times \mathcal{E}(s_1)$$

- The client takes the requested data s_i by decrypting a.

[2]In general, $q = (\mathcal{E}(0), \cdots, \mathcal{E}(1), \cdots, \mathcal{E}(0))$, where only ith element is $\mathcal{E}(1)$ and others are $\mathcal{E}(0)$.

How this works? Let us analyze for each i, either 0 or 1. If i is 0, then $q = (\mathcal{E}(1), \mathcal{E}(0))$, and

$$\begin{aligned} a &= \mathcal{E}(1) \times \mathcal{E}(s_0) + \mathcal{E}(0) \times \mathcal{E}(s_1) \\ &\equiv \mathcal{E}(1 \times s_0 + 0 \times s_1) && \text{(by homomorphism)} \\ &\equiv \mathcal{E}(s_0) \end{aligned}$$

therefore, $\mathcal{E}^{-1}(a) = s_0$. The remaining case with $i = 1$ is similar. Intuition behind this protocol is:

- Each 0 and 1 is a token representing validity: 1 means valid, and 0 means not.
- A request q' is a tuple consisting of the (encrypted) validity tokens: ith element is $\underline{1}$ (meaning that the very element is valid), and others are $\underline{0}$ (meaning that the elements are not valid).
- A server, given request $q = (q_0, q_1)$, does not know which one is an encryption of valid token, but he can generate an answer by masking (multiplying) each data s_i with validity token q_i. In this case, non valid data is always masked by $\underline{0}$, and valid data is only masked by $\underline{1}$. Therefore, only valid data remains as it is, others becoming $\underline{0}$. At this moment, summing all masked data yields the very result, s_i.

Using this concept, we can execute *if* statement without noticing any information which branch to be taken. For example, as for the following *if* expression,

$$\textit{if}\ (b)\ \{x := 10\}\ \textit{else}\ \{x := 20\}$$

suppose that b can have either 0 or 1 during execution. Secret execution evaluates $\underline{b}$ to either $\underline{0}$ or $\underline{1}$. In this case, executor has no idea of $\underline{b}$'s value, but he can evaluate x's value after *if* statement. How? We know that x's value is either 10 or 20 according to the value of b: if b is 1 then x becomes 10, and if b is 0 then x becomes 20. Suppose that we have equality testing function $\underline{eq}$ returning $\underline{1}$ if two encrypted values are equivalent, otherwise $\underline{0}$. (Refer to Section 4.2 for details.) Then secret executor can evaluate x's value as follows:

$$\begin{aligned} \underline{x} &= \underline{eq}(\mathcal{E}(b), \mathcal{E}(1)) \times \mathcal{E}(10) + \underline{eq}(\mathcal{E}(b), \mathcal{E}(0)) \times \mathcal{E}(20) \\ &= \mathcal{E}(eq(b, 1)) \times \mathcal{E}(10) + \mathcal{E}(eq(b, 0)) \times \mathcal{E}(20) && \text{(by homomorphism)} \\ &= \mathcal{E}(eq(b, 1) \times 10 + eq(b, 0) \times 20) && \text{(by homomorphism)} \end{aligned}$$

If b is $\mathcal{E}(1)$ then the above value becomes $\mathcal{E}(10)$, else if b is $\mathcal{E}(0)$ then the above value becomes $\mathcal{E}(20)$. Here, the executor has no idea of either what is b's value or which branch to be taken, but correctly evaluates x's value.

This approach of masking each validity token and summing up all, called "masking and sum", is a central, seemingly magical, device making it possible for executor to evaluate program without noticing any information about given program.

2 Preliminaries

2.1 Notations

Suppose an homomorphic encryption scheme $\mathcal{E}$, a tuple of (`KeyGen`, `Encrypt`, `Decrypt`, `Evaluate`). Given a security parameter λ, let $(pk, sk) \leftarrow \texttt{KeyGen}(1^\lambda)$. For the simplicity, we write $\mathcal{E}(m)$ for the ciphertext c such that $c \leftarrow \texttt{Encrypt}(pk, m)$ for any plaintext m, and we write $\mathcal{E}^{-1}(c)$ for the plaintext m such that $m \leftarrow \texttt{Decrypt}(sk, c)$ for any ciphertext c. We also write $\underline{m}$ for $\mathcal{E}(m)$. We say that two ciphertexts are equivalent if their decryption results are equal: $c \equiv c'$ if $c \leftarrow \mathsf{Encrypt}(pk, m)$ and $c' \leftarrow \mathsf{Encrypt}(pk, m)$.

2.2 Semantic Security

An encryption scheme is said to be semantically secure[13], if, roughly speaking, given a ciphertext c that encrypts either m_0 or m_1, any adversary cannot decide which one is encrypted, even if the adversary chooses m_0 and m_1, and furthermore their encrypted results c_0 and c_1 are provided. Semantic security implies that the underlying encryption scheme should be probabilistic: given plaintext, there must be many ciphertext candidates, among which encryption algorithm should choose one randomly according to certain distribution. Therefore, the probability that c is equal to either c_0 or c_1 is negligibly closed to zero.[3] If an encryption scheme is deterministic, it cannot be semantically secure because an adversary can easily decide by comparing c and c_i's.

2.3 Fully Homomorphic Encryption Scheme

An encryption scheme is said to be *fully* homomorphic if the encryption scheme preserves arbitrary number of addition (modular 2) and multiplication operations, i.e., for plaintexts m_1, m_2,

$$\begin{aligned}\mathcal{E}(m_1) + \mathcal{E}(m_2) &\equiv \mathcal{E}(m_1 + m_2)\\ \mathcal{E}(m_1) \times \mathcal{E}(m_2) &\equiv \mathcal{E}(m_1 \times m_2)\end{aligned}$$

One such possible encryption scheme is proposed by Gentry[7] and Brakerski and Vaikuntanathan[1].

3 Programs

We present our target language, a simple imperative language. It is simple yet powerful enough to represent fundamental machine instructions: load, store, arithmetic operations, and conditional jump. Machine code level representation is quite adequate because a program should eventually be compiled into machine code in order to be executed.

Our target programs are given in the form of graph. For the sake of simplicity, we focus on simple, integer arithmetic programs without complicated data structures such as pointers or arrays. Note that it is straightforward extension to support those data structures.

$$\begin{array}{rl}
\textit{Programs} & P = G\\
\textit{Control Flow Graphs} & G = \langle N, E, \psi_N, \psi_E \rangle\\
\textit{Nodes} & N\\
\textit{Edges} & E \in \textit{Nodes} \times \textit{Nodes}\\
\textit{Basic Blocks} & \psi_N \in \textit{Nodes} \to \textit{Statements}^*\\
\textit{Guards} & \psi_E \in \textit{Edges} \to \textit{Conditions}\\
 & \\
\textit{Statements} & s ::= x := e\\
\textit{Atomic Expressions} & i ::= x \mid c\\
\textit{Expressions} & e ::= i + i \mid i \times i\\
\textit{Conditions} & b ::= e \mid \neg e\\
\textit{Variables} & x\\
\textit{Constants} & c \in \mathbb{N}
\end{array}$$

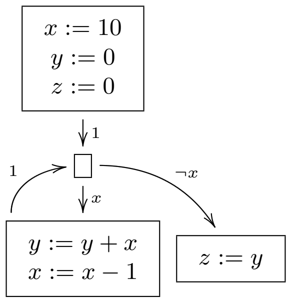
A program is a graph whose nodes represent basic blocks and edges represent conditional expressions. A basic block is a series of statements; here only a single type of statement exists: assign statement. A conditional expression that lies on a edge indicates condition for the edge

[3]Less than an inverse of any polynomial in the security parameter, i.e., $< 1/2^\lambda$. We call such quantity just *negligible*.

to be valid; a control flow is only established when the corresponding conditional expression is evaluated to be true. Also, an expression consists of only atomic expressions, for the sake of simplicity.

Example 1. *The following example presents c-like program (left-hand side) and their graph representation (right-hand side). Each edge has conditional expression. Edges with conditional expression* 1 *means that the edges are always valid.*

```
x := 10
y := 0
z := 0
while (x) {
    y := y + x
    x := x - 1
}
z := y
```



3.1 Program Execution

Execution of our target programs is defined in the usual way. Program execution is a sequence of states, a transitive closure of transition relations. A transition corresponds to each step of execution. A state represent current node and memory.

$$\begin{array}{rrl} \textit{Transitions} & \hookrightarrow & \in \textit{States} \times \textit{States} \\ \textit{States} & S & \in \textit{Nodes} \times \textit{Memories} \\ \textit{Memories} & M & \in \textit{Variables} \xrightarrow{\texttt{fin}} \textit{Constants} \end{array}$$

$$\begin{array}{rrl} \textit{Evaluations} & \phi_s^* & \in (\textit{Memories} \times \textit{Statements}^*) \rightarrow \textit{Memories} \\ & \phi_s & \in (\textit{Memories} \times \textit{Statements}) \rightarrow \textit{Memories} \\ & \phi_e & \in (\textit{Memories} \times \textit{Expression}) \rightarrow \textit{Constants} \\ & \phi_b & \in (\textit{Memories} \times \textit{Conditions}) \rightarrow \{0,1\} \\ & \phi_x & \in (\textit{Memories} \times \textit{Variables}) \rightarrow \textit{Constants} \end{array}$$

Evaluation algorithms for each syntactic categories are defined in the usual way. A transition relation is defined as follows:

$$\begin{array}{rl} (N,M) \hookrightarrow (N',M') \iff & (N,N') \in \textit{Edges} \quad \textit{and} \\ & \phi_b(M, \psi_E(N,N')) = 1 \quad \textit{and} \\ & \phi_s^*(M, \psi_N(N')) = M' \end{array}$$

Example 2. *For the following simple sequential program:*

$$n_1 : \boxed{x := 1} \xrightarrow{1} n_2 : \boxed{x := 2} \xrightarrow{1} n_3 : \boxed{x := 3}$$

its execution is as follows:

$$(n_1, \{x \mapsto 1\}) \hookrightarrow (n_2, \{x \mapsto 2\}) \hookrightarrow (n_3, \{x \mapsto 3\})$$

We only consider a deterministic program, that is, given a state, the number of its next states cannot be more than one.

Definition 1 (Deterministic Programs)**.** *A program is said to be deterministic if:*

$$\begin{array}{l} (N,M) \hookrightarrow (N',M') \quad \textit{and} \quad (N,M) \hookrightarrow (N'',M'') \\ \qquad\qquad \implies N' = N'' \quad \textit{and} \quad M' = M'' \end{array}$$

4 Secret Execution Protocol: Statements

Secret execution protocol is a tuple of algorithms: key generation, encryption, decryption, and execution. The key generation algorithm is induced from the base encryption scheme, and the decryption algorithm is naturally induced from the encryption algorithm. Thus, here we will focus encryption and execution algorithm.

We first explain algorithms for *statements*, specifically how to handle variable assignments, variable lookups, and arithmetic operations. After this, we will explain how to handle basic blocks, and finally, entire control flow graphs, in Section 5.

4.1 Encryption

Base Encryption Scheme Our secret execution protocol is based on homomorphic encryption scheme $\mathcal{E}$ on binary bits: the message space is $\{0, 1\}$. The base encryption scheme $\mathcal{E}$ is required to be: semantically secure, public key algorithm, and fully homomorphic. One such possible encryption scheme is proposed by Gentry[22].

Constants Using the base encryption scheme, we can construct encryption algorithm for constants, $\mathcal{E}_c$, by encrypting each bit of the given constant and making it as a tuple. For some constant c, its encryption algorithm is defined as follows:

$$\mathcal{E}_c(c) \overset{\texttt{def}}{=} (\mathcal{E}(c_1), \cdots, \mathcal{E}(c_n)) \quad \textit{where } c = (c_1 \cdots c_n)_2$$

where n is a predefined value that is big enough for all constants appeared at a given program (e.g., $n = 32$ for all programs on 32-bit machine).

Variables Encryption algorithm for variables, $\mathcal{E}_x$, maps a given variable to some constant using predefined mapping table ϕ, followed by encrypting the resulted constant using $\mathcal{E}_c$:

$$\mathcal{E}_x(x) \overset{\texttt{def}}{=} \mathcal{E}_c(\phi(x)) \quad \textit{given } \phi : \textit{Variables} \to \mathbb{N}$$

Note that this encryption algorithm is still secure, even if the mapping table ϕ is revealed, because the base encryption scheme is semantically secure.

Atomic Expressions Given an atomic expression i, encryption algorithm for atomic expressions, $\mathcal{E}_i$, generates a tuple whose first element is an encrypted result of type of i, and second element is an encrypted result of i itself:

$$\begin{aligned} \mathcal{E}_i(x) &\overset{\texttt{def}}{=} (\mathcal{E}_{op}(\texttt{VAR}), \mathcal{E}_x(x)) \\ \mathcal{E}_i(c) &\overset{\texttt{def}}{=} (\mathcal{E}_{op}(\texttt{CON}), \mathcal{E}_c(c)) \end{aligned}$$

Expressions Given an expression e, encryption algorithm for expressions, $\mathcal{E}_e$, generates a tuple whose first element is an encrypted result of operation's type (i.e., op-code) of e, and second (and third) element is an encrypted result of first (and second, resp.) operand of e:

$$\begin{aligned} \mathcal{E}_e(i_1 + i_2) &\overset{\texttt{def}}{=} (\mathcal{E}_{op}(\texttt{ADD}), \mathcal{E}_i(i_1), \mathcal{E}_i(i_2)) \\ \mathcal{E}_e(i_1 \times i_2) &\overset{\texttt{def}}{=} (\mathcal{E}_{op}(\texttt{MUL}), \mathcal{E}_i(i_1), \mathcal{E}_i(i_2)) \end{aligned}$$

Conditions Encryption algorithm for conditional expressions, $\mathcal{E}_b$, is defined in a similar way with $\mathcal{E}_e$:

$$\begin{aligned}\mathcal{E}_b(e) &= (\mathcal{E}_{op}(\texttt{NOP}), \mathcal{E}_e(e))\\ \mathcal{E}_b(\neg e) &= (\mathcal{E}_{op}(\texttt{NEG}), \mathcal{E}_e(e))\end{aligned}$$

Note that the dummy op-code, `NOP`, is inserted at the first case. This is because such dummy op-code contributes to hide size information of original expression; otherwise, an attacker can easily recognize the type of original expression from the size of the encrypted expression—the longer encrypted conditional expressions, the more likely to be of type `NEG`.

Statements Encryption algorithm of statements, $\mathcal{E}_s$, simply generates a tuple consisting of encrypted results of each part of assignment statement:

$$\mathcal{E}_s(x := e) \overset{\texttt{def}}{=} (\mathcal{E}_x(x), \mathcal{E}_e(e))$$

Operation Codes Encryption algorithm of operation codes, $\mathcal{E}_{op}$, is similar with the encryption algorithm of variables, $\mathcal{E}_x$: mapping each operation code to some constant, followed by encrypting the constant using $\mathcal{E}_c$.

Example 3. *For the following statement:*

$$x := 1 + 2$$

the encrypted result is as follows: (suppose that $\phi(x) = 0$ and $\phi_{op}(\texttt{ADD}) = 3$ and $\phi_{op}(\texttt{CON}) = 0$)

$$\begin{aligned}&\mathcal{E}_s(x := 1 + 2)\\ =&(\mathcal{E}_x(x), (\mathcal{E}_{op}(\texttt{ADD}), (\mathcal{E}_{op}(\texttt{CON}), \mathcal{E}_c(1)), (\mathcal{E}_{op}(\texttt{CON}), \mathcal{E}_c(2))))\\ =&(\mathcal{E}_c(0), (\mathcal{E}_c(3), (\mathcal{E}_c(0), \mathcal{E}_c(1)), (\mathcal{E}_c(0), \mathcal{E}_c(2))))\\ =&(\mathcal{E}_c(00_2), (\mathcal{E}_c(11_2), (\mathcal{E}_c(00_2), \mathcal{E}_c(01_2)), (\mathcal{E}_c(00_2), \mathcal{E}_c(10_2))))\\ =&((\mathcal{E}(0), \mathcal{E}(0)), ((\mathcal{E}(1), \mathcal{E}(1)), ((\mathcal{E}(0), \mathcal{E}(0)), (\mathcal{E}(0), \mathcal{E}(1))),\\ &\qquad((\mathcal{E}(0), \mathcal{E}(0)), (\mathcal{E}(1), \mathcal{E}(0)))))\end{aligned}$$

4.2 Basic Tools for Execution

Before directly diving into execution of encrypted statements, it will be helpful to introduce basic tools to be used in secret execution.

Equality Test First, we need to figure out how to test equality between two encrypted data. Conventional equality testing functions, *eq* and *neq*, are defined as follows: (for the sake of simplicity, we consider 1 as true, and 0 as false.)

$$eq(x, y) \overset{\texttt{def}}{=} \begin{cases} 1 & \textit{if } x = y \\ 0 & \textit{if } x \neq y \end{cases} \qquad neq(x, y) \overset{\texttt{def}}{=} \begin{cases} 0 & \textit{if } x = y \\ 1 & \textit{if } x \neq y \end{cases}$$

What we have to do is defining equality testing functions on encrypted data, $\underline{eq}$ and $\underline{neq}$, preserving original semantics of *eq* and *neq*, such that:

$$\underline{eq}(\mathcal{E}(x), \mathcal{E}(y)) \equiv \mathcal{E}(eq(x, y))$$

Let's first consider the case that x and y is one bit constant—either 0 or 1. In this case, we can easily find out *neq* has exactly same behavior with XOR gate, which can be simulated by addition (modulo 2) operation. Therefore, we can define $\underline{eq}$ and $\underline{neq}$ for bit as follows: (where $\underline{1}$ is an encrypted value of 1)

$$\underline{eq}(\underline{x}, \underline{y}) \overset{\texttt{def}}{=} \underline{x} + \underline{y} + \underline{1}$$
$$\underline{neq}(\underline{x}, \underline{y}) \overset{\texttt{def}}{=} \underline{x} + \underline{y}$$

The above definition is correct since *eq* and *neq* for bits are defined as follows,[4]

$$\begin{aligned} eq(x,y) &= NOT(XOR(x,y)) &= x +_2 y +_2 1 \\ neq(x,y) &= \quad XOR(x,y) &= x +_2 y \end{aligned}$$

and the base encryption scheme $\mathcal{E}$ is homomorphic, preserving addition and multiplication (modulo 2) operations.

Furthermore, we can also test equality between arbitrary bits value, by $\underline{or}$'ing all primitive equality testing results between each bits from two values, as follows:[5] (where $\underline{x_i}$ and $\underline{y_i}$ are ith elements of $\underline{\vec{x}}$ and $\underline{\vec{y}}$ respectively)

$$\underline{\vec{eq}}(\underline{\vec{x}}, \underline{\vec{y}}) \overset{\texttt{def}}{=} \underline{n\vec{eq}}(\underline{\vec{x}}, \underline{\vec{y}}) + \underline{1}$$
$$\underline{n\vec{eq}}(\underline{\vec{x}}, \underline{\vec{y}}) \overset{\texttt{def}}{=} \underline{or}(\cdots \underline{or}(\underline{neq}(\underline{x_1}, \underline{y_1}), \underline{neq}(\underline{x_2}, \underline{y_2}))\cdots, \underline{neq}(\underline{x_n}, \underline{y_n}))$$

Note that encrypted result of multi-bits constant is represented by a tuple, such as $\underline{\vec{x}}$ and $\underline{\vec{y}}$ in the above definition. Refer to encryption of constants, $\mathcal{E}_c$, for details.

Case-matching We can now define case-matching operation, the most fundamental operation for program execution. A simple form of case-matching operation, *ifelse*, is defined as follows: (for the sake of simplicity, we assume that all values—x, y, v, and w—are integer constants)

$$ifelse(x,y,v,w) \overset{\texttt{def}}{=} \begin{cases} v & \text{if } x = y \\ w & \text{if } x \neq y \end{cases}$$

We can define case-matching operation for encrypted values, $\underline{ifelse}$, as follows:[6]

$$\underline{ifelse}(\underline{x}, \underline{y}, \underline{v}, \underline{w}) \overset{\texttt{def}}{=} (\underline{eq}(\underline{x}, \underline{y}) \times \underline{v}) + (\underline{neq}(\underline{x}, \underline{y}) \times \underline{w})$$

In the above definition, if $\underline{x}$ is equal to $\underline{y}$,[7] then $\underline{eq}(\underline{x}, \underline{y})$ becomes $\underline{1}$, which makes $(\underline{eq}(\underline{x}, \underline{y}) \times \underline{v})$ to be $\underline{v}$,[8] and $\underline{neq}(\underline{x}, \underline{y})$ becomes $\underline{0}$, which makes $(\underline{neq}(\underline{x}, \underline{y}) \times \underline{w})$ to be $\underline{0}$; summing up the two results, eventually, yields $\underline{v}$. The remaining case in which $\underline{x}$ is not equal to $\underline{y}$ is similar. Therefore, $\underline{ifelse}$ preserves the semantics of *ifelse* function, as follows:

$$\underline{ifelse}(\mathcal{E}(x), \mathcal{E}(y), \mathcal{E}(v), \mathcal{E}(w)) \equiv \mathcal{E}(ifelse(x,y,v,w))$$

We can also define case-matching operation $\underline{case}$, a simple extension of $\underline{ifelse}$, as follows:

$$\underline{case}(\underline{x}, (\underline{c_i}, \underline{v_i})_i) \overset{\texttt{def}}{=} \begin{array}{l} (\underline{eq}(\underline{x}, \underline{c_1}) \times \underline{v_1}) \ + \\ (\underline{eq}(\underline{x}, \underline{c_2}) \times \underline{v_2}) \ + \\ \qquad \vdots \\ (\underline{eq}(\underline{x}, \underline{c_n}) \times \underline{v_n}) \end{array}$$

[4]Note that logic gates can be simulated by addition (modulo 2) and multiplication operations, with considering 1 as true and 0 as false, such as: $XOR(x,y) = x + y \pmod 2$, $AND(x,y) = x \times y$, and $NOT(x) = x + 1 \pmod 2$.

[5]Operator $\underline{or}$ can be defined as follows: $\underline{or}(\underline{x}, \underline{y}) \overset{\texttt{def}}{=} (\underline{x} + \underline{y}) + (\underline{x} \times \underline{y})$.

[6]We implicitly extend the notion of addition and multiplication for bits to one for tuples in a point-wise fashion, as follows: $x \times (y_1, \cdots, y_n) = (x \times y_1, \cdots, x \times y_n)$, and $(x_1, \cdots, x_n) + (y_1, \cdots, y_n) = (x_1 + y_1, \cdots, x_n + y_n)$.

[7]Strictly speaking, $\underline{x}$ is equivalent to $\underline{y}$, that is, two values are encryptions for same original value.

[8]Strictly speaking, another encryption of v

where $case(x, \{(c_1, v_1), \cdots, (c_n, v_n)\})$ means that if x is equal to c_1 then result value is v_1, or if x is equal to c_2 then result is v_2, and so on.

4.3 Execution

Now we present how to execute encrypted statements, using the basic tools presented in Section 4.2. Secret execution algorithm for statements, $\underline{\phi_s}$, consists of sub-algorithms for each syntactic categories: $\underline{\phi_i}$, $\underline{\phi_e}$, $\underline{\phi_b}$, and $\underline{\phi_x}$. We first present how to execute encrypted op-codes—$\underline{\phi_i}$, $\underline{\phi_e}$, and $\underline{\phi_b}$—and then how to execute memory operations under encryption—$\underline{\phi_x}$ and $\underline{\phi_s}$.

Atomic Expressions An encrypted atomic expression is a tuple whose first element is type of operation—opcode—and second is its operand. In order to execute an atomic expression, we firstly identify its opcode, and carry out certain operation according to the opcode. However, this is not the case in execution of an encrypted expression; we have no idea of the opcode—which is encrypted. Then, how can we execute expressions without identifying their opcodes? We only have to use the case-matching function $\underline{case}$.

We can define execution algorithm for encrypted atomic expressions, $\underline{\phi_i}$, as follows:

$$\begin{aligned}\underline{\phi_i}(\underline{M}, (\underline{\texttt{OP}}, \underline{i})) = \underline{case}(\underline{\texttt{OP}}, \{&(\underline{\texttt{VAR}}, \underline{\phi_x}(\underline{M}, \underline{i})),\\ &(\underline{\texttt{CON}}, \underline{i})\})\end{aligned}$$

Given an input memory $\underline{M}$, an opcode $\underline{\texttt{OP}}$, and an operand $\underline{i}$, function $\underline{\phi_i}$ evaluates resulting value—integer constant. Since secret executor cannot identify which value $\underline{\texttt{OP}}$ has (among $\underline{\texttt{VAR}}$, $\underline{\texttt{CON}}$), secret executor first carries out all possible operations, and asks the $\underline{case}$ function to select appropriate one among those candidates. Comparing $\underline{\texttt{OP}}$ with each possible operators, the case-matching function $\underline{case}$ multiplies each comparison results with corresponding candidate values, followed by summing up all, which leaves only one value—an encrypted result of execution. Note that secret executor cannot notice anything about which candidate was selected. Rather, secret executor just carries out always same series of operations regardless of its input, yet the execution result is properly generated by itself.

Expressions We can also define execution algorithm for encrypted expressions, $\underline{\phi_e}$, similarly with $\underline{\phi_i}$, as follows:[9]

$$\begin{aligned}&\underline{\phi_e}(\underline{M}, (\underline{\texttt{OP}}, \underline{i_1}, \underline{i_2}))\\ &\quad = \underline{case}(\underline{\texttt{OP}}, \{(\underline{\texttt{ADD}}, ADD(\underline{\phi_i}(\underline{M}, \underline{i_1}), \underline{\phi_i}(\underline{M}, \underline{i_2}))),\\ &\qquad\qquad\qquad (\underline{\texttt{MUL}}, MUL(\underline{\phi_i}(\underline{M}, \underline{i_1}), \underline{\phi_i}(\underline{M}, \underline{i_2})))\})\end{aligned}$$

Conditions We can also define secret execution algorithm for conditional expressions, $\underline{\phi_b}$, similarly with the above, as follows:[10]

$$\begin{aligned}\underline{\phi_b}(\underline{M}, \underline{\texttt{OP}}, \underline{e}) = \underline{case}(\underline{\texttt{OP}}, \{&(\underline{\texttt{NOP}}, \vec{\underline{or}}(\underline{\phi_e}(\underline{e}))),\\ &(\underline{\texttt{NEG}}, \vec{\underline{or}}(\underline{\phi_e}(\underline{e})) + \underline{1})\})\end{aligned}$$

Memory Lookups In secret execution, variable lookup need to be carried out without identifying which variable to be lookuped; thus, secret executor fetches all entries from given memory, and asks $\underline{case}$ to select proper one. Secret execution for variable lookups, $\underline{\phi_x}$, is defined as follows:

$$\underline{\phi_x}(\underline{M}, \underline{x}) = \underline{case}(\underline{x}, \{(\underline{x'}, \underline{M}(\underline{x'})) \mid \underline{x'} \in Dom(\underline{M})\})$$

[9] Operators ADD and MUL are logical circuits constructed by using XOR and AND gates.

[10] $\vec{\underline{or}}(\vec{\underline{x}}) \overset{\text{def}}{=} \underline{or}(\cdots \underline{or}(\underline{x_1}, \underline{x_2}) \cdots, \underline{x_n})$ where $\vec{\underline{x}} = (\underline{x_1}, \cdots, \underline{x_n})$.

Assignments Similarly, secret execution for assignments updates every entries, where newly updated value is selected between the given value and previously stored value, by $\underline{\mathit{ifelse}}$ function.

$$\begin{aligned}&\underline{\phi_s}(\underline{M}, \underline{x}, \underline{e})\\ &\quad= \{\underline{x'} \mapsto \underline{\mathit{ifelse}}(\underline{x}, \underline{x'}, \underline{\phi_e}(\underline{M}, \underline{e}), \underline{M}(\underline{x'})) \mid \underline{x'} \in \mathit{Dom}(\underline{M})\}\end{aligned}$$

Problem of this approach is that secret executor cannot insert a new entry, that is, number of entries of memory does not change during execution. Therefore, initial memory $\underline{M_0}$ should have all entries to be used for execution in advance. It is possible to enumerate all entries of memory in advance of execution, since the number of all variables appeared in given program is predetermined, say N. We can construct initial memory state $\underline{M_0}$ as follows:[11]

$$\underline{M_0} = \{\mathcal{E}_c(i) \mapsto \mathcal{E}_c(0) \mid i \in [1, N]\}$$

Note that all secret executor have to know is just N, total number of variables, not the whole set of variables itself. We can even hide the number of variables by providing secret executor a number greater than N. By doing so, we can prevent revealing any information about variables except their maximum size.

5 Secret Execution Protocol: CFGs

Now we explain how to encrypt and execute basic blocks and entire control flow graphs.

5.1 Basic Blocks

Encryption We can easily encrypt a basic block by encrypting each statements of the basic block.

The problem, however, is that the number of statements vary with each basic blocks. One can distinguish very big basic block from small one, which provides some hints to an attacker.

Thus, we need to fix the number of statements of basic blocks, say n. This fixed number n can be set to maximum number of statements among all basic blocks, or set to average number of statements—in this case, one should split basic blocks who is bigger than the average.

As for the basic blocks who is smaller than the fixed number n, we insert dummy statements. A dummy statement is marked with validity token 0, and an original one with 1, by which we can distinguish which one is dummy. Of course, the validity tokens are encrypted so that secret executor cannot find out which one is genuine. Nevertheless, secret execution yields same result with one of executing only original statements. For example,

$$\begin{array}{lcl} x_1 := e_1; & & (\mathcal{E}_x(x_1), \mathcal{E}_e(e_1), \mathcal{E}_c(1)); \\ & & (\mathcal{E}_x(x'_1), \mathcal{E}_e(e'_1), \mathcal{E}_c(0)); \\ x_2 := e_2; & \stackrel{\mathcal{E}}{\Longrightarrow} & (\mathcal{E}_x(x_2), \mathcal{E}_e(e_2), \mathcal{E}_c(1)); \\ & & (\mathcal{E}_x(x'_2), \mathcal{E}_e(e'_2), \mathcal{E}_c(0)); \\ x_3 := e_3; & & (\mathcal{E}_x(x_3), \mathcal{E}_e(e_3), \mathcal{E}_c(1)); \end{array}$$

Using this approach, we can make all basic blocks to have same size, which makes it impossible for attackers to distinguish basic blocks by their size, preventing statistical attack fundamentally/ultimately.

[11]In order to do so, we need to make the variable mapping function, ϕ, to map each variables to integer constant in order starting from 1.

Execution Given a sequence of statements, we can easily extend the notion of ϕ_s to ϕ_s^*. Execution algorithm for sequence of statements ϕ_s^* evaluates each statement in turn, passing output memory state of current statement to next statement:

$$\underline{\phi_s^*}(\underline{M}, (\underline{s_1}; \underline{s_2}; \cdots; \underline{s_n})) = \underline{\phi_s}(\cdots \underline{\phi_s}(\underline{\phi_s}(\underline{M}, \underline{s_1}), \underline{s_2}) \cdots, \underline{s_n})$$

In this case, each execution of statement asks $\underline{\mathit{ifelse}}$ function to select proper behavior: updating results or bypassing current statement.[12]

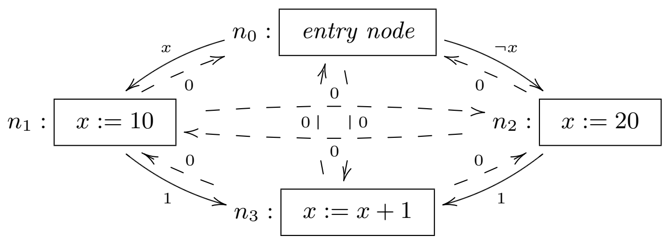
$$\begin{aligned}\underline{\phi_s'}(\underline{M}, \underline{x}, \underline{e}, \underline{\alpha}) = \underline{\mathit{case}}(&\underline{\alpha}, (\underline{0}, \underline{M}),\\ &(\underline{1}, \underline{\phi_s}(\underline{M}, \underline{x}, \underline{e})))\end{aligned}$$

5.2 Control Flow Graphs

Encryption We can also easily encrypt a control flow graph by encrypting contents of each nodes (i.e., basic blocks) and edges (i.e., conditional expressions).

However, this encryption method cannot hide a structure of the given control flow graph. An attacker can easily find the number of nodes and edges, and also which nodes are connected with.

Thus, we need to insert additional dummy nodes and dummy edges to build fully connected graph, which hides not only the number of nodes but also entire graph's structure—control flows. For example, we can hide the graph structure by inserting dummy edges as follows: (dotted lines mean dummy edges)



Execution Secret program execution is a sequence of encrypted states, a transitive closure of encrypted transition relations. An encrypted transition relation, $\underline{\hookrightarrow}$, corresponds to each step of secret execution. An encrypted state, $\underline{\mathit{States}}$, represents an encrypted memory of currently executed node, and also, encrypted memories of other nodes which are results of most recent execution of each nodes. Thus, an encrypted state is given by a map from nodes to encrypted memories associated with encrypted activity-bit which indicates whether the associated node is currently executed (i.e., active) or not. (Note that *underlined* are objects of encrypted domain. *Nodes* is not encrypted.)

$$\begin{array}{rrl}
\underline{\mathit{Transitions}} & \underline{\hookrightarrow} & \in \underline{\mathit{States}} \times \underline{\mathit{States}} \\
\underline{\mathit{States}} & \underline{S} & \in \mathit{Nodes} \xrightarrow{\texttt{fin}} (\underline{\mathit{Memories}} \times \underline{\mathit{Activities}}) \\
\underline{\mathit{Memories}} & \underline{M} & \in \underline{\mathit{Variables}} \xrightarrow{\texttt{fin}} \underline{\mathit{Constants}} \\
\underline{\mathit{Activities}} & \underline{\alpha} & \in \underline{\{0,1\}} \\
\\
\underline{\mathit{Evaluations}} & \underline{\phi_s^*} & \in (\underline{\mathit{Memories}} \times \underline{\mathit{Statements}^*}) \to \underline{\mathit{Memories}} \\
& \underline{\phi_s} & \in (\underline{\mathit{Memories}} \times \underline{\mathit{Statements}}) \to \underline{\mathit{Memories}} \\
& \underline{\phi_e} & \in (\underline{\mathit{Memories}} \times \underline{\mathit{Expression}}) \to \underline{\mathit{Constants}} \\
& \underline{\phi_b} & \in (\underline{\mathit{Memories}} \times \underline{\mathit{Conditions}}) \to \underline{\{0,1\}} \\
& \underline{\phi_x} & \in (\underline{\mathit{Memories}} \times \underline{\mathit{Variables}}) \to \underline{\mathit{Constants}}
\end{array}$$

[12]Here, *bypassing* does not mean ignoring execution of the statement, instead, meaning that it executes the statement but immediately abandons the resulting memory state. Note that if a cryptographic system allows bypassing statements' execution literally, then the system will not be secure any more.

- Objectives: Given an encrypted state $\underline{S}$, output $\underline{S}'$ such that $\underline{S} \underline{\hookrightarrow} \underline{S}'$.
- Algorithm: For each node $n \in \mathit{Nodes}$, compute $(\underline{M}', \underline{\alpha}')$ such that $\underline{S}'(n) = (\underline{M}', \underline{\alpha}')$.
  - Let $\underline{S}(n) = (\underline{M}, \underline{\alpha})$.
  - For each in-edge $(n_i, n) \in \mathit{Edges}$, compute $\underline{\alpha}'_i$ and $\underline{M}'_i$ such that:
    - Let $\underline{S}(n_i) = (\underline{M}_i, \underline{\alpha}_i)$.
    - $\underline{\alpha}'_i \leftarrow \underline{\phi_b}(\underline{M}_i, \psi_E(n_i, n)) \times \underline{\alpha}_i$
    - $\underline{M}'_i \leftarrow \underline{M}_i \times \underline{\alpha}'_i$.
  - $\underline{\alpha}' \leftarrow \bigvee \underline{\alpha}'_i$.
  - $\underline{M}_{in} \leftarrow \Sigma \underline{M}'_i$.
  - $\underline{M}_{out} \leftarrow \underline{\phi^*_s}(\underline{M}_{in}, \psi_N(n))$.
  - $\underline{M}' = \underline{\mathit{ifelse}}(\underline{\alpha}', \underline{1}, \underline{M}_{out}, \underline{M})$.

Figure 1: Algorithm of encrypted transition relation $\underline{\hookrightarrow}$

An algorithm of encrypted transition relation is defined in Figure 1. Unlike the original program execution at Section 3.1, the secret execution algorithm knows neither which node to be executed (i.e., active) nor which control flow to be taken. Thus, this algorithm also use same principle: "masking and sum". In this case, the activity bit plays a role of a validity bit. Coupled with a conditional expression, an activity bit guides the algorithm to take appropriate control flow. In this way, the execution algorithm selectively updates only an active node, even if it executes all nodes. For example, execution of the above program is follows: (for the simplicity, we present values in unencrypted form.)

n_0	$\{\}$, 1	$\{\}$, 0	$\{\}$, 0	$\{\}$, 0
n_1	$\{\}$, 0	$\{x \mapsto 10\}, 1$	$\{x \mapsto 10\}, 0$	$\{x \mapsto 10\}, 0$
n_2	$\{\}$, 0	$\{\}$, 0	$\{\}$, 0	$\{\}$, 0
n_3	$\{\}$, 0	$\{\}$, 0	$\{x \mapsto 11\}, 1$	$\{x \mapsto 11\}, 0$

Each column represents a state: a tuple of a memory and an activity bit. Note that starting from the entry node n_0, activity bit 1 is moved along execution, and is gone when execution is finished (in the last column). Once all activity bits become 0, further execution does not update anything. Also, note that only one single node has activity bit of $\underline{1}$ during execution, if we initially have made only starting node's activity bit to be $\underline{1}$.[13]

By the way, is it possible for secret executor to determine when to stop repeating atomic execution? The answer is No. The stopping point should be given from external observer—program's owner. If secret executor can determine when to stop, the system is not secure, revealing critical information of given program which might be used in statistical attack. Therefore, secret executor periodically asks the program's owner to check whether program's execution is terminated or not. Secret executor *or*'ing all activity bits and send it back to owner, and the owner decrypts the value and checks whether it is 0 or not—0 means that program's execution is terminated.

6 Properties

Now we present some properties of our secret execution protocol.

[13]More precisely, the starting node should not equal to entry node of encrypted graph—a graph with additional dummy nodes—where the starting node means actual entry node of original graph.

6.1 Client Privacy

Our secret execution protocol $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Exec})$ provides client privacy, in that the client's program is kept to be semantically secure, without reference to the Exec algorithm. (Indeed Exec is a public algorithm with no secrets.)

Lemma 1 (Client privacy)**.** *The secret execution protocol, a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Exec})$*, described by Section 4 and Section 5, provides client privacy, that is,*

- *The advantage of the adversary* Adv *in the following game is negligible in the security parameter* λ*:*
  - Adv *chooses two programs* p_0 *and* p_1*, where* $|p_0| = |p_1|$*.*
  - *Let* $b \leftarrow \{0, 1\}$*,* $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$*, and* $q \leftarrow \mathsf{Enc}(pk, p_b)$*.*
  - Adv *is given* (pk, q) *and outputs* b'*.*
- *The advantage of* Adv *is* $|Pr[b' = b] - 1/2|$*.*

Proof. By the semantic security of the base encryption scheme. □

6.2 Correctness

Our secret execution protocol preserves original execution, in that each step of secret execution simulates each step of original execution. Before presenting simulation lemma, we need to define state encryption and decryption.

Definition 2 (States Encryption & Decryption)**.** *Let* $\underline{S_0} = \{N \mapsto (\underline{M_0}, \underline{0}) \mid N \in \mathit{Nodes}\}$.[14] *For any state* $S = (N, M)$*, state encryption* $\mathcal{E}_S$ *is defined as follows:*

$$\mathcal{E}_S(S) = \underline{S_0} + \{N \mapsto (\underline{M}, \underline{1})\}$$

Also, state decryption $\mathcal{E}_S^{-1}$ *is defined as follows:*

$$\mathcal{E}_S^{-1}(\underline{S}) = (N, \mathcal{E}^{-1}(\underline{M})) \iff \underline{S}(N) = (\underline{M}, \underline{\alpha}) \quad \textit{and} \quad \mathcal{E}^{-1}(\underline{\alpha}) = 1$$

Using state encryption/decryption, we can state simulation lemma as follows:

Lemma 2 (Simulation)**.** *For any state* S*, let* $S \hookrightarrow S'$*,* $\underline{S} = \mathcal{E}_S(S)$*, and* $\underline{S} \underline{\hookrightarrow} \underline{S'}$*. Then* $\mathcal{E}_S^{-1}(\underline{S'}) = S'$*.*

$$S \hookrightarrow S' \quad \Longrightarrow \quad \begin{tikzcd} S \arrow[d, "\mathcal{E}_S"'] & S' \\ \underline{S} \arrow[r, hook, "\underline{}"'] & \underline{S'} \arrow[u, "\mathcal{E}_S^{-1}"'] \end{tikzcd}$$

Proof. By the homomorphism of the base encryption scheme. □

6.3 Security Overhead

Security overhead is quadratic of program size, more specifically $O(V \times N)$, where V is the number of variables and N is the number of nodes of the given program. Overhead comes from memory operations and execution of CFGs. Each memory operation, either lookup or assignment, needs to examine all entries of the memory, while normal memory operation does not. Also, execution of CFGs (Figure 1) needs to examine all nodes of the graph, while normal execution evaluates a single node. Finally, the computational overhead of Gentry's fully homomorphic encryption scheme[7] is quasi-linear of λ^9.

[14]Refer to Section 4.3 for the definition of $\underline{M_0}$.

7 Feasibility Discussion

7.1 Somewhat Homomorphic Encryption Scheme

It would be more practical if our secret execution protocol is based on somewhat homomorphic encryption scheme.

Currently, all *fully* homomorphic encryption scheme—basis of our protocol—are still in "proofs of concept" stage. Here, "fully" means that arbitrary many number of addition and multiplication is preserved. Although many improvements actively has been made[18, 2, 10, 1, 11, 21, 17, 9], it is still too heavy/expensive to be practical.

On the other hand, there exist several *somewhat* homomorphic encryption scheme[1, 2][15] that are already quite practical[17]. They, however, preserve only limited number of addition and multiplication, especially more sensitive to multiplication—AND gate.

In order to adopt somewhat homomorphic encryption scheme, we need to reduce multiplication depth of our protocol. Multiplication depth is defined as the number of nested multiplication, which is a similar concept of "depth of circuit"—the number of nested AND gates of a given circuit. Currently, our protocol's multiplication depth is $O(n)$, where n is the number of bits of ciphertext for atomic expressions (i.e., variables or constants), which is $O(\mathit{word\_size} \times \mathit{security\_parameter})$. This multiplication depth is too high for state-of-the-art somewhat homomorphic encryption scheme to afford. They are currently practical up to dozens of depth[18].

One possible way to reduce multiplication depth is to use homomorphic encryption scheme that supports operations in $\mathbb{Z}_p$ as well as $\mathbb{Z}_2$—our base encryption scheme. In that way, we can reduce our protocol's multiplication depth to $O(1)$. This is because in $\mathbb{Z}_p$ we can conduct addition and multiplication on $\log p$ bits. For example, for $p \geq 2^{32}$, multiplication depth of 32-bit machine (programs) is $O(1)$.

7.2 Partially Homomorphic Encryption Scheme

We can also make our protocol to be more practical by using *partially* homomorphic encryption scheme as an base encryption scheme. Here, "partial" means that the encryption scheme preserves only a single operation: either addition or multiplication. There already exists many practical partially homomorphic encryption scheme.

We can implement the "masking and sum" method, a central key concept of our protocol (mentioned at Section 1.1), using just *partially* homomorphic encryption scheme. Suppose that $\mathcal{E}$ is a partially homomorphic encryption scheme that preserves only addition operation.

$$\mathcal{E}(m_1) + \mathcal{E}(m_2) = \mathcal{E}(m_1 + m_2)$$

Then, multiplication of two ciphertexts is derived as follows:

$$\begin{aligned}
\mathcal{E}(m_1) \times \mathcal{E}(m_2) &= \underbrace{\mathcal{E}(m_1) + \cdots + \mathcal{E}(m_1)}_{\mathcal{E}(m_2)} \\
&= \mathcal{E}(\underbrace{m_1 + \cdots + m_1}_{\mathcal{E}(m_2)}) \\
&= \mathcal{E}(m_1 \times \mathcal{E}(m_2))
\end{aligned}$$

Note that the derived term is doubly encrypted. Based on the above equation, we can have

[15]Actually, fully homomorphic encryption scheme is based on somewhat homomorphic encryption scheme. An universal technique, so-called bootstrapping[7], make a given somewhat homomorphic encryption scheme to be fully homomorphic.

the "masking and sum" method as follows:

$$\begin{aligned}\mathcal{E}(0) \times \mathcal{E}(m_0) + \mathcal{E}(1) \times \mathcal{E}(m_1) &= \mathcal{E}(0 \times \mathcal{E}(m_0)) + \mathcal{E}(1 \times \mathcal{E}(m_1)) \\ &= \mathcal{E}(0) + \mathcal{E}(\mathcal{E}(m_1)) \\ &= \mathcal{E}(0 + \mathcal{E}(m_1)) \\ &= \mathcal{E}(\mathcal{E}(m_1))\end{aligned}$$

Here, $\mathcal{E}(0)$ and $\mathcal{E}(1)$ are validity tokens. By masking each validity token and summing up all, we can choose between $\mathcal{E}(m_0)$ and $\mathcal{E}(m_1)$—here, $\mathcal{E}(m_1)$ is selected. The above method can be described more generally as follows:

$$\Sigma(\mathcal{E}(a_i) \times \mathcal{E}(m_i)) = \mathcal{E}(\mathcal{E}(m_k)) \quad \textit{where } a_i = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

Note that this method is based on just *partially* homomorphic encryption scheme: it preserves only addition operation.

The problem is that the result is doubly encrypted—$\mathcal{E}(\mathcal{E}(m))$; one need to decrypt it twice. If the "masking and sum" methods are nested with depth n, one need to decrypt the result $n + 1$ times, which leads to increase overhead of the client. This overhead, however, can be reduced by using such *partially* homomorphic encryption scheme that its decryption algorithm is very efficient and low-cost.

Another problem is that we have not yet found a way to implement $\underline{\vec{eq}}$ algorithm using *partially* homomorphic encryption scheme, as well as the case of *somewhat* homomorphic encryption scheme mentioned at Section 7.1.

7.3 Partial Secret Execution

As an practical use, we can apply our secret execution protocol to only a part of program. It would be useful to secretly execute a part of program that is very important for security, such as a routine of checking serial key for genuine software, or a routine checking consistency of program to see if it is not falsified. In this case, such routine is very small part of a given program, so that secret execution of such part is affordable.

The problem is that secret execution yields an encrypted result, which is supposed to be used by other part of program via normal execution. One possible solution is to communicate with a server who is able to decrypt the result. For example, program's consistency checking routine is secretly executed and yields encrypted result: either $\underline{true}$ or $\underline{false}$, then the result is sent to a central server who is supposed to decrypt it and send it back to the program. In this way, one can totally hide the underlying algorithm of consistency checking routine; otherwise, malicious users can freely tamper with a program with the knowledge of consistency checking algorithm.

8 Applications

8.1 One-time Execution

One of possible application of secret execution protocol is one-time execution. One-time execution is an execution strategy in which a given program is executed only once, and spoiled so that the program cannot be executed again. One-time execution would be useful in case a server sends a client a program to be executed only once, and never be executed again. For example, let's imagine a financial consulting firm who has a software that given one's financial standing, find optimal asset management. Suppose that customers do not want to expose their financial standing, and the company want to protect their software from being freely used. In this case, one promising solution is that the company gives a customer their software that is one-time executable, and the customer run the software with their private information.

Our secret execution protocol enables one-time execution by constructing secret executor for secret executor. More specifically,

- First, we design a special virtual machine that executes a given program and immediately destroys the program. Note that this virtual machine is also a kind of program.
- Next, we write the target program which can be executed on the virtual machine.
- Finally, we encrypt both the virtual machine and the target program, and secretly execute the encrypted virtual machine.

In this way, we can hide not only the target program, but also the virtual machine (i.e., one-time executor), so that malicious users cannot modify the virtual machine in order to put aside the target program.

8.2 Crash Report

Another possible application is secure crash report. You are often asked to send crash report when you are using a computer, but you do not because the crash report may contain private information. In this case, secret execution can be a good solution: you can send the crash report after encrypting it, and a receiver can examine the encrypted crash report without decrypting it. For example, a developer makes an inspection program which analyzes a given crash report, and encrypts the inspection program. Then, the encrypted program can be secretly executed with the encrypted crash report, and yields an encrypted inspection result.

9 Related Work

Code Obfuscation Code obfuscation [19, 12] is formalized by means of abstract interpretation [5]. Attackers to code obfuscation are modeled as abstract interpreters. Potency of code obfuscation, which means that “the obfuscated program is harder to understand than the original one” [19, 12], is measured by comparing the most concrete preserved property or incompleteness of abstract interpretation. In this framework, the attackers in the lattice of abstract semantics. Note that we can compare abstract semantics by the amount of information they contain and they form a lattice of which the bottom is the concrete semantics and the top is the trivial semantics.

Code obfuscation aims to hide information of the program by a program transformation. On the other hand, secret execution aims to hide information of input and output in addition to that of program. Furthermore, one who hides information of the program and one who hides information of the input and the output may be different in secret execution. Code obfuscation requires the target language and the source language be the same and the obfuscated program behaves observationally equal to the original program. On the other hand, secret execution only guarantees that the plain output be recoverable from the encrypted output by decryption.

Their framework does not deal with the resilience of obfuscation, which means that measuring the difficulty of breaking the obfuscation is not properly addressed. On the other hand, attackers to secret execution are modeled as cryptographic attackers and both potency and resilience are addressed in the cryptographic settings. Cryptographically, Semantic security of secret execution implies the resilience of it in one of the best ways we can hope for. We think it is a proper way to address the resilience.

Branching Program on Encrypted Data Ishai and Paskin proposed a protocol for evaluating *length-bounded branching programs* on encrypted data [16]. Given a branching program P and an encryption c of an input x, their protocol produces a cipher text c', an encryption of $P(x)$. The protocol is based on oblivious transfer and partially homomorphic public-key encryption scheme.

Our secret execution supports more general features with respect to computation and security. While their protocol only handles length-bounded branching programs, our secret execution deals with arbitrary programs which may contain loops. Also we supports storage so that load and store operations are computable as well as purely mathematical functions. Furthermore, secret execution hides the program itself in addition to the input and output.

References

[1] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 505–524, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *IN PRINCIPLES OF PROGRAMMING LANGUAGES 1998, POPL98*, pages 184–196, 1998.

[4] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering*, 28:735–746, 2002.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[6] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985.

[7] C. Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[8] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[9] C. Gentry and S. Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *FOCS*, pages 107–109, 2011.

[10] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, EUROCRYPT'11, pages 129–148, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *CRYPTO*, pages 850–867, 2012.

[12] R. Giacobazzi and I. Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *SAS*, pages 129–145, 2012.

[13] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[14] J. R. Gosler. Software protection: Myth or reality? In *Advances in Cryptology*, CRYPTO '85, pages 140–157, London, UK, UK, 1986. Springer-Verlag.

[15] K. Heffner and C. Collberg. The obfuscation executive. In K. Zhang and Y. Zheng, editors, *Information Security*, volume 3225 of *Lecture Notes in Computer Science*, pages 428–440. Springer Berlin Heidelberg, 2004.

[16] Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Proceedings of the 4th conference on Theory of cryptography*, TCC'07, pages 575–594, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? Technical Report MSR-TR-2011-61, Microsoft Research, 2011.

[18] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[19] M. D. Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[20] M. O. Rabin. How to exchange secrets with oblivious transfer. Technical Report TR-81, Aiken Computation Lab., Harvard University, 1981.

[21] N. Smart and F. Vercauteren. Fully homomorphic simd operations. Cryptology ePrint Archive, Report 2011/133, 2011.

[22] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the 29th Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, pages 24–43, Berlin, Heidelberg, 2010. Springer-Verlag.