

Secret Shared Random Access Machine

Shlomi Dolev and Yin Li^(✉)

Department of Computer Science,
Ben-Gurion University of the Negev, Beersheba, Israel
`dolev@cs.bgu.ac.il`, `yunfeiyangli@gmail.com`

Abstract. The computations over RAM are preferred over computations with circuits or Turing machines. Secure and private RAM executions become more and more important in the scope avoiding information leakage when executing programs over a single computer as well as over the clouds. In this paper, we propose a distributed scheme for evaluating RAM programs without revealing any information on the computation including the program, the data and the result. We use the Shamir secret sharing to share all the program instructions and private string matching technique to ensure the correct instruction execution. We stress that our scheme obtains information theoretic security and does not rely on any computational hardness assumptions, therefore, gaining indefinite private and secure RAM execution of perfectly unrevealed programs.

Keywords: Shamir secret sharing · Random access machine · Information theoretic secure

1 Introduction

Cloud computing provides cost-efficient and flexible shared infrastructure and computational services on demand for various customers who need to store and operate on a huge amount of data. Until now, there are various services providers such as Amazon [1] and Google [13] offering platforms, software, and storage outsourcing applications. Much attention has been paid to them due to the potential benefits and business opportunities that clouds could bring [9].

However, cloud computing also introduces security and privacy risks for the clients. For example, some of the cloud providers are not perfectly reliable and are vulnerable to network attacks and data leakage. Furthermore, even a single computer with the same cloud organization is untrustworthy. There are possible attacks on a single computer during which information is copied from the bus of the computer and sent to an adversary.

Several techniques are applied to address data storage privacy [18–20, 26] and security computation on clouds [17, 29]. Among these studies, security

S. Dolev—Partially supported by Kamin grant of the Israeli economy ministry, and the Rita Altura Trust Chair in Computer Sciences.

Y. Li—The author would like to acknowledge the Lynne and William Frankel Center as it supports students travel for presenting their works.

in evaluating random access machine (RAM) program is an important task [2, 23], since many modern algorithms are operating on the von Neumann RAM architecture. Until now, there are mainly two ways, the first is to convert a RAM program into circuits and the second is to use oblivious RAM, introduced by Goldreich and Ostrovsky [19]. Oblivious RAM schemes are preferred as there is no need to convert the program into a binary circuit which leads to a huge blowup in program size and its running time.

Even though the propositions for secure RAM evaluation can address various privacy challenges including two-party [22, 23], multiparty [5, 10] or large-scale computation [6] against semi-honest or malicious adversaries, they all assume that the processors used by clouds are trustworthy. Thus, in these proposals, the CPU has to decrypt the input data before processing and then encrypt the output data again. In fact, an adversary can introduce a special hardware Trojan [28] designed to disable or destroy a system in the future, or leak confidential information. Similar attack has already been demonstrated in [3], where a specially designed Trojan in the CPU revealed sensitive information to the adversary.

Threat Model. We assume that there is a client that wants to run a program on the clouds. But the client does not want to reveal any information about both the program and the data. The adversary, has deployed the untrusted hardware to the clouds. That is to say, the adversary can listen to the bus, may extract information on the internal activity of the processor. All the clouds are not necessarily semi-honest.

Unfortunately, none of the above protocols can avoid information leakage under such threat model. Thus, one may wish to execute an encrypted program on encrypted data without decrypting neither the program nor the data. A straightforward approach is to execute the encrypted instructions in the clouds processors directly. Fully homomorphic encryption [14, 15] (FHE) is a way to achieve this goal. Several schemes are proposed to implement secret program execution over FHE (e.g., [7, 8, 31]). However, the main problem is that the proposed schemes have high overhead of computation [16] which make FHE more theoretical result than practical. Moreover, Gentry's scheme and later FHE schemes relied on the hardness assumptions such that of the ideal lattices, which are only computationally secure, rather than key-less information theoretical secure.

Our Contribution. In this paper, an alternative architecture is proposed with security and privacy that are based on theoretical security promises. The main technique is a combination of Shamir Secret Sharing [25] and the recently proposed Accumulating Automata [12].

Secret sharing is used to utilize perfect privacy of the client's program and processor states and secret string matching [12] is used to facilitate instruction execution. We note that the modern instruction set, for example, CISC and RISC, originally designed for efficiency and performance [21], are too complicated when there is a need to hide their nature of operation and the sequence of operations they form. Thus we apply One Instruction Set Computer (OISC) to

our model. We simulate the OISC instruction *subtract and branch if less than or equal to zero* (Subleq) that is proven complete and for which there exists a compiler from high-level programming languages to Subleq [24]. As a result, our scheme has the following significant characteristics

- *Information theoretic security.* We use Shamir secret sharing which could provide information theoretic security for clients. In our scheme, the user’s program is secret shared and run on independent machines and clouds. Each cloud only needs to perform computation without communicating with other clouds. Moreover, note that we use the instruction Subleq proven to be complete in terms of Turing-complete computation. Thus, our model can execute any RAM programs privately and securely.
- *Program protection.* During the whole execution of the program, for every instruction, the processors have to “touch” all the instructions in the memory. Moreover, for every data access, the processors also have to access all of the data items. The execution mode and access pattern make the client program “oblivious” to the clouds, thus ensuring both data and program privacy. Still, the operations can be delegated by the users to powerful machines in the clouds, which perform these linear access to all items for executing operations without revealing their nature and sequence.
- *Error correcting.* Notice that the clients run their programs in E independent machines/clouds. According to the conclusion of Ben-Or et al. [4], as long as less than one-third of clouds are malicious (do not follow the protocol possibly returning wrong results), the client can detect their actions by reconstructing the final result using Lagrange interpolation.

The rest of the paper is organized as follows: in Sect. 2, we briefly introduce the settings used in our paper. Section 3 describes the basic primitives we use in our construction. Explicit application and its security analysis are given in Sect. 4. Finally, conclusions are drawn in Sect. 5.

2 Preliminary

In this section, we briefly introduce the basic ingredients used in the sequel.

Shamir Secret Sharing. Shamir secret sharing (SSS) is an information theoretic secure protocol, which allows a dealer to secret share a values s among E players. There is a threshold δ for the scheme, such that, the knowledge of δ or fewer player secrets make the adversary learn no information about s , but if more than δ players communicate their shares to each other, they can easily recover the secret.

Distribution: The dealer picks a random polynomial $f \in \mathbb{F}_p[x]$ of degree $\delta < E$ such that $f(0) = s \in \mathbb{F}_p$. The dealer also chooses E arbitrary non-zero indices $\alpha_1, \dots, \alpha_E$, computes $f(\alpha_i)$ for $1 \leq i \leq E$ and send $(\alpha_i, f(\alpha_i))$ to each corresponding players.

Reconstruction: Any $\delta + 1$ players can reconstruct the polynomial f by applying Lagrange interpolation to the tuples $(\alpha_i, f(\alpha_i))$. They recover the secret by computing $f(0) \bmod p = s$.

Shamir secret sharing is additively homomorphic but is not multiplicatively homomorphic. Namely, if we want to perform multiplication using Shamir secret shares, a special “degree reduction step” is required. We will discuss this problem more explicitly in the following section.

Private String Matching. Recently, Dolev et al. proposed a secret string matching algorithm using Accumulating Automata [12]. The algorithm runs on several cloud servers. The strings to be compared are originally secret shared using Shamir secret sharing and therefore stay unknown to the processing servers. Note that the comparison of two strings represented in secret shares is different from the comparison of strings in a plaintext format, as each participant cannot judge the compare result independently.

Unary representation: The authors of [12] demonstrated their scheme over unary letter representation, where each letter is represented by a binary number with hamming weight 1. For example, letter a – z are expressed by the binary strings: $a = [100 \cdots 00]$, $b = [010 \cdots 00]$, $c = [001 \cdots 00]$, \dots , $z = [000 \cdots 01]$ with each representation consists of 26 bits. We use the expression $S = \sum_{i=0}^r u_i \times v_i$, to compare two letters, where $[u_0 u_1 \cdots u_r]$ and $[v_0 v_1 \cdots v_r]$ are two unary representations. It is clear that whenever the two representations are identical, S is equal to 1, otherwise S is equal to 0. Assume that each cloud has the secret shares of these two representations, i.e., $(\alpha, f_i(\alpha))$ and $(\alpha, g_i(\alpha))$, where $f_i(0) = u_i$ and $g_i(0) = v_i$. Similarly, it can compute the following equation to identify whether the two letters are identical:

$$\sum_{i=1}^r (f_i(\alpha) \times g_i(\alpha)). \quad (1)$$

We have following lemma.

Lemma 1. *If the two letters are identical, then the result of Eq. (1) is the secret share of 1, otherwise the result of this equation is a secret share of 0.*

Proof. Note that u_i, v_i are the secret bit and would be either 1 or 0. Let $f'_i(\alpha)$ and $g'_i(\alpha)$ denote the evaluation of $f(x)$ and $g(x)$ at point α without the constant term u_i, v_i , respectively. We can see

$$\begin{aligned} & f_i(\alpha) \times g_i(\alpha) \\ &= (f'_i(\alpha) + u_i) \times (g'_i(\alpha) + v_i) \\ &= f'_i(\alpha)g'_i(\alpha) + u_i g'_i(\alpha) + v_i f'_i(\alpha) + u_i v_i \\ &= F(\alpha) + u_i v_i, \end{aligned}$$

where $F(\alpha) = f'_i(\alpha)g'_i(\alpha) + u_i g'_i(\alpha) + v_i f'_i(\alpha)$. Therefore, $f_i(\alpha) \times g_i(\alpha)$ can be seen as a secret share of $u_i v_i$. It is clear that only when $u_i = v_i = 1$, $f_i(\alpha) \times g_i(\alpha)$ is a secret share of 1, and otherwise it is a secret share of 0. Note that the hamming weight of unary representation is only 1, one can directly find the final summation is at most 1 which conclude the result.

Based on this observation, it is easy to compare a string using Accumulating Automata, which is a type of finite automata. Only when the string letters are exactly the same, the last node will be set to 1, otherwise this node will stay 0. One can reconstruct the values of this node to identify whether the string matching is successful or not.

Binary representation: The main drawback of unary representation is that it has too many redundant bits. For example if we want to represent the numbers 1 to 1000, we have to use 1000 bits. An alternative method is to use binary representation.

Assume that there are two letters represented as $[u_0u_1 \cdots u_r]_2$ and $[v_0v_1 \cdots v_r]_2$, where $u_i, v_i \in \{0,1\}$. We compare these letters using the Algorithm 1.

As a simple example, we consider two binary strings $[1010]_2$ and $[1101]_2$. According to previous description, we perform the following computations:

- Bitwise subtraction,
 $[1, 0, 1, 0] - [1, 1, 0, 1] = [1 - 1, 0 - 1, 1 - 0, 0 - 1] = [0, -1, 1, -1];$
- Bitwise squaring,
 $[0^2, (-1)^2, 1^2, (-1)^2] = [0, 1, 1, 1];$
- Bitwise OR, $S = 0|1|1|1 = 1;$ ¹

Algorithm 1. Secret comparison using binary representation

```

1: for  $i = 1$  to  $r$  do
2:    $s_i = [u_i - v_i]^2$ 
3: end for
4:  $S = 0$ 
5: for  $i = 1$  to  $r$  do
6:    $S = S + s_i - S \times s_i$ 
7: end for
8: return  $1 - S$ 

```

It is easy to check that if the two strings are equal, S is equal to 0 and otherwise to 1. In this example, the value of S is 1. In order to return the same value as the unary representation, we prefer to return $1 - S$ rather than S . Note that we only use the subtraction/addition and multiplication in the above algorithm, similarly to the unary case, these operations can also be implemented using Shamir secret sharing. However, compared with unary representation, it requires either more participants or (more) degree reduction operations.

One Instruction Computer Set. OISC is an abstract machine that uses only one instruction. It is proven that OISC is capable of being a universal computer in the same manner as traditional computers with multiple instructions [24]. This indicates that one instruction set computers are very powerful despite the simplicity of the design, and can achieve high throughput under certain configurations.

Since there is only one instruction in the system, it needs no identification to determine which instruction to execute. Thus, we only need to design the implementation of one instruction. Actually, there are several options for choosing the OISC instruction, such as *subtract and branch if not equal to zero* (SBNZ), *subtract and branch if less than or equal to zero* (Subleq), *add and branch unless positive* (Addleq). Among these instructions, Subleq is the most commonly used. Nowadays, there are Subleq compiler and Subleq-based processor [27] which

¹ One can check that Step 6 in Algorithm 1 is equivalent to the bitwise OR operation.

make Subleq a practical and efficient choice. Therefore, in this paper, we focus on how to simulate Subleq privately and secretly. Comparing the values of two memory words that are represented by secret shares, is hard to implement, hence we secret share the words bit by bit, perform the arithmetic over secret shared bits and then branch according to the sign bit of the result. This leads to a novel scheme for executing secret shared Subleq (SSS-Subleq) programs. The details are presented in Sect. 3.

3 SSS-Subleq Programs and Their Execution

Since our architecture is built on Subleq, for any client programs written by high-level languages, it needs to be compiled into Subleq codes at first [27]. Then the client executes the set of Subleqs over the system. In the following, we will investigate the implementation details of Subleq using Shamir secret sharing.

The SSS-Subleq Format and Architecture Overview. According to the definition of Subleq, it has three parameters A, B, C where the contents at address B are subtracted from the contents at address A , and the result is stored at address B , and then, if the result is not greater than 0, the execution jumps to the memory address C , otherwise it continues to the next instruction in the sequence. The pseudo code is given in the procedure $\text{Subleq}(A, B, C)$. Here, the PC (program counter) is a pointer that indicates the address of next instruction.

Note that the Subleq contains some important operations: load, store, subtraction and conditional branch. Thus, in order to execute Subleq using Shamir secret sharing, we have to simulate the following operations using secret shares:

Procedure. $\text{SUBLEQ}(A, B, C)$

```

1:  $\text{Mem}[B] = \text{Mem}[B] - \text{Mem}[A]$ 
2: if  $\text{Mem}[B] \leq 0$  then
3:   goto  $C$ 
4: else
5:   goto  $PC + 1$ 
6: end if
```

- $\text{LOAD}(H)$: Load the instruction in address H to the processor.
- $\text{JUMP}(C)$: Transfers control to index C , implement the branching operation.
- $\text{READ}(X)$: Read the data at address X .
- $\text{WRITE}(X, Y)$: Write the data Y in address X .

Please note that the operation $\text{goto } PC + 1$ and $\text{goto } C$ can be implemented by the operation JUMP with different parameters. Among all these operations, a critical problem is how to find the right address secretly. Fortunately, secret string matching allows us to implement these operations without revealing any information. According to the description in Sect. 2, we use unary representation to represent the addresses including memory addresses and instruction indices where each bit of the unary representation is encoded as a secret shared value. The format of the SSS-Subleq instruction has five parts which are shown in Fig. 1.

The first block stores the instruction index number which is equivalent to the instruction address, the second and third blocks store the operand addresses and the fourth to fifth blocks store the branch index C and the index of next instruction, respectively.

index	A	B	C	$PC + 1$
-------	-----	-----	-----	----------

Fig. 1. Format of SSS-Subleq

Besides the former operations, there is a need to implement the subtraction between two operands and determine the next instruction address according to the subtraction result. Therefore, we choose to represent every operand as a signed number. In order to perform subtraction in an easy way, we use two's complement representation where subtraction can be transformed into addition. The most significant bit (MSB) is the sign bit. Analogous with the address, each bit of the operands is secret shared. The outline of our RAM architecture is presented Fig. 2. In our architecture, we use a modified Harvard architecture which not only physically separates storage and signal pathways for instructions and data, but also separates the read-only and read/write part of data. Note that since Shamir secret sharing is not multiplicatively homomorphic, degree reduction is needed after several multiplications. This special structure allows us to implement read and write operations in relatively efficient manner. In particular, the degree of the polynomials used for the read-only part (possibly big-data corpus) is unchanged throughout the execution(s).

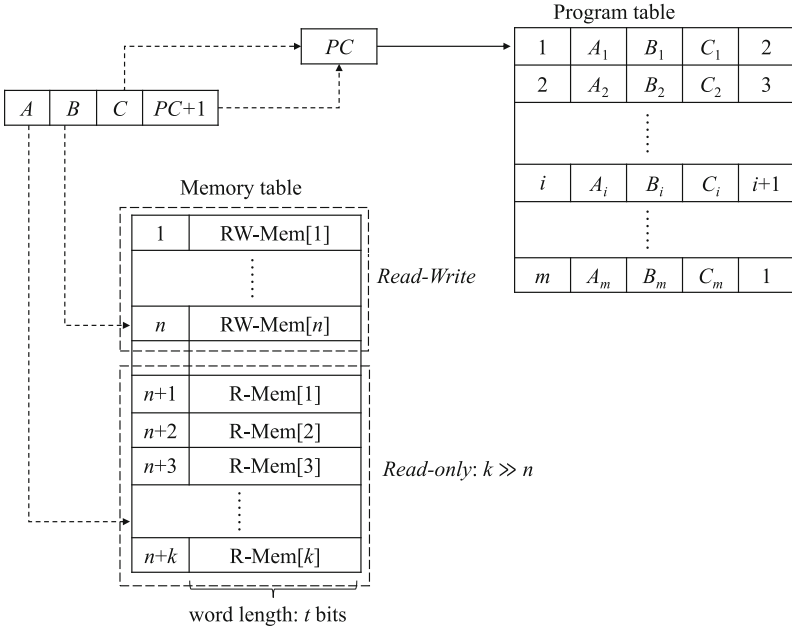


Fig. 2. Architecture

Table 1. The parameters of a program

Parameter	Description
m	The number of instructions of the user program
n	The number of data items that can be accessed for read and write
k	The number of data items that can be accessed for read only
t	The bit length of the data stored in the memory

The parameters of our architecture are presented in Table 1. Here, we assume that the client program reads a large number of data items compared with the data items the program writes to, thus we have $k \gg n$. In the following, we will show how to simulate the four basic operations using the Accumulating Automata technique.

Operation Details. We start describing the implementation of a function called: $compare(U, V, r)$, where U and V are secret shares of the unary address consisting of r elements. For example, let $U = u_1, u_2, \dots, u_r$, $V = v_1, v_2, \dots, v_r$ denote the secret shares of two such parameters, we compute

$$compare(U, V, r) = \sum_{i=1}^r (u_i \times v_i) \quad (2)$$

According to Sect. 2, the above expression testifies whether U, V are identical or not. It is easy to check that the result of $compare(U, V, r)$ is a secret share of 1 if $U = V$, and otherwise, if $U \neq V$, is 0.

Now we describe the details of the four operations:

Description of LOAD: The initial values of S_i are set to 0, and the symbol \parallel means concatenation of all values from S_1 to S_4 . H represents the secret shares of the instruction address which we want to load and η_i represents secret shares of the i -th instruction address. It is clear that the value returned is the right instruction we want to load.

Description of READ: According to Fig. 2, the format of memory table consists of two parts: the address number ϵ_i and data θ_i . Analogous to the corresponding analysis for the LOAD operation, we can easily check that S is the data whose index number is equal to X .

Procedure. LOAD(H)

```

1: for  $i = 1$  to  $m$  do
2:    $Num_i \leftarrow compare(H, \eta_i, m)$ 
3:    $S_1 \leftarrow S_1 + Num_i \times A_i$ 
4:    $S_2 \leftarrow S_2 + Num_i \times B_i$ 
5:    $S_3 \leftarrow S_3 + Num_i \times C_i$ 
6:    $S_4 \leftarrow S_4 + Num_i \times (PC_i + 1)$ 
7: end for
8: return  $S_1 \parallel S_2 \parallel S_3 \parallel S_4$ 

```

Procedure. READ(X)

```

1: for  $i = 1$  to  $n + k$  do
2:    $Num_i \leftarrow compare(X, \epsilon_i, n + k)$ 
3:    $S \leftarrow S + Num_i \times \theta_i$ 
4: end for
5: return  $S$ 

```

Description of WRITE: The operation implements writing the data Y in the address X using secret shares. Note that only when ϵ_i equals X , the Num_i is the secret shares of 1, and then the data Y can substitute the former data item, otherwise the data will not be changed.

Procedure. WRITE(X, Y)

```

1: for  $i = 1$  to  $n$  do
2:    $Num_i \leftarrow compare(X, \epsilon_i, n + k)$ 
3:    $\theta_i \leftarrow \theta_i + Num_i \times (Y - \theta_i)$ 
4: end for

```

Description of JUMP: The operation JUMP is nearly the same as the operation LOAD. If the program needs to execute the C -th instruction in the program table, it just assigns the last part of current instruction to the PC . Then the program will “jump” to the destination.

Procedure. JUMP(C)

```

1:  $PC \leftarrow C$ 
2: LOAD( $PC$ )

```

Implementation of SSS-Subleq. We then investigate the conditional branch that required in Subleq in secret shares form. It is difficult to compare two numbers directly since all the numbers are secret shared and the clouds never know the secrets. Here, we use two’s complement to represent the operands and using the sign bit to implement the comparison. In two’s complement, the sign bit of positive integer is 0 and negative integer is 1. Therefore, when implementing SUBLEQ(A, B, C), we can use the sign bit of $Mem[B] - Mem[A]$ to (blindly) decide whether to branch or not. The only problem is that the integer 0, for which the sign bit in its representation is also 0, while it should imply branching. This problem can be fixed by a slight modification: using the sign bit of $Mem[B] - Mem[A] - 1$ instead of sign bit of $Mem[B] - Mem[A]$. Moreover, we will show that this sign bit can be obtained during the computation of $Mem[B] - Mem[A]$ in the following paragraphs.

Two’s Complement Subtraction. The advantage of using two’s complement is the elimination of examining the signs of the operands to determine if addition or subtraction is needed. Therefore, to compute subtraction $\beta - \alpha$, it only need to perform following steps:

- Convert α : Invert every bit of α and add one, denoted by $\bar{\alpha} + 1$.
- Addition: Perform binary addition and discard any overflowing bit, denoted by $\beta + \bar{\alpha} + 1$.

Note that we also need the sign bit of $\beta - \alpha - 1$. As described above, using two’s complement representation, the subtraction $\beta - \alpha$ is converted to $\beta + \bar{\alpha} + 1$. Similarly, the subtraction $\beta - \alpha - 1$ is implemented as

$$\underline{\beta - \alpha - 1} = \underline{\beta + \bar{\alpha} + 1} - 1 = \beta + \bar{\alpha}.$$

The similarity allows us to implement these two subtractions together.

The algorithm for two’s complement subtraction using Shamir secret sharing is given in Algorithm 2. According to previous description in Sect. 2, we know

Algorithm 2. The two's complement subtraction using Shamir secret sharing

```

1: procedure SSS-SUB( $A, B$ )
2:   Input:  $A = [a_{t-1}a_{t-2} \cdots a_1a_0]$ ,  $B = [b_{t-1}b_{t-2} \cdots b_1b_0]$  where  $a_i, b_i$  are secret
   shares of bits of two's complement represented number.
3:   Output:  $R = [r_{t-1}r_{t-2} \cdots r_1r_0]$  where  $R = B - A$ , and the sign bit of  $B - A - 1$ 
4:    $a_0 = 1 - a_0$  ▷ Invert of the least significant bit
5:    $carry[0] = a_0 \cdot b_0$ 
6:    $r_0 = a_0 + b_0 - 2 \cdot carry[0]$  ▷ Addition of the least significant bit
7:   for  $i = 1$  to  $t - 1$  do
8:      $a_i = 1 - a_i$  ▷ invert each bit  $A \rightarrow \bar{A}$ 
9:      $r_i = a_i + b_i - 2a_ib_i$ 
10:     $carry[i] = a_ib_i + carry[i - 1] \cdot r_i$  ▷ The carry bit
11:     $r_i = r_i + carry[i - 1] - 2 \cdot carry[i - 1] \cdot r_i$  ▷ The result bit
12:  end for
13:   $sign = r_{t-1}$  ▷ The sign bit of  $B - A - 1$ , used for branch
14:   $carry[0] = r_0$  ▷ Add 1 to the result obtain  $B - A$ 
15:   $r_0 = 1 - r_0$ 
16:  for  $i = 1$  to  $t - 1$  do
17:     $carry[i] = r_i \cdot carry[i - 1]$ 
18:     $r_i = r_i + carry[i - 1] - 2 \cdot carry[i]$ 
19:  end for
20:  return ( $R || sign$ )
21: end procedure

```

Algorithm 3. The Shamir secret sharing based Subleq

```

1: procedure SSS-SUBLEQ( $A, B, C$ )
2:    $R_1 \leftarrow \text{READ}(A)$ 
3:    $R_2 \leftarrow \text{READ}(B)$ 
4:    $R || Num = \text{SSS-SUB}(R_1, R_2)$ 
5:    $\text{WRITE}(B, R)$ 
6:    $\text{JUMP}(Num \cdot C + (1 - Num) \cdot (PC + 1))$ 
7: end procedure

```

the multiplications and additions/subtractions of the shares correspond to those of the secrets. Thus one can easily check that Algorithm 2 implements the two's complement subtraction.

Therefore, Subleq can be implemented with secret shares by Algorithm 3. In step 6, we can check that if the value represented by R_2 is less than R_1 , then $Num = 1, PC = C$, else $Num = 0, PC = PC + 1$. Therefore, this expression implement the conditional branch of Subleq.

Degree Reduction. The main bottleneck of our scheme is the multiplication with shares used in the basic operations, as the Shamir secret sharing is not multiplication homomorphic. Note that multiplying two polynomials gives a polynomial with a degree that is equal to the sum of the degrees of the source polynomials. We observe that the READ, JUMP and LOAD increase the polynomial degrees related to each secret shared bit stored in the registers, the subtraction

and WRITE increase the degrees related to the data items stored in the memory. Therefore, we have to process the degree reduction for these data items at a certain time. In [11], Dolev et al. proposed a method for reducing the polynomial degree without revealing the original secret. In our model, we define a *reducer* that is in charge of reducing the polynomial degrees and a *randomizer* in charge of generating random polynomials for all the participants. Note that the codes of the *reducer* and the *randomizer* should be executed independently in order to protect the secret s , but either of them can be executed by the dealer machine. The polynomial degree reduction algorithm appears in Algorithm 4.²

Algorithm 4. Polynomial degree reduction for secret shares

```

1: procedure DECREASE( $P(x), d, d^*$ )
2:   Let  $u_1, \dots, u_E$  be  $E$  participants,  $D$  be the randomizer and  $R$  be the reducer.
3:   Let  $P(x) \in \mathbb{F}_p[x]$  of degree  $d$  is the polynomial for secret  $s$ .
4:    $D$  randomly selects polynomial  $f(x)$  of degree  $d$  and  $g(x)$  of degree  $d^*$ , where
       $f(x)$  and  $g(x)$  have the same constant term.
5:   for  $i = 1$  to  $E$  do
6:      $D$  sends  $(f(u_i), g(u_i))$  to  $u_i$ .
7:      $u_i$  computes  $P(u_i) + f(u_i)$  and sends it to  $R$ .
8:   end for
9:    $R$  interpolates and computes a polynomial  $Q(x) = P(x) + f(x)$ .
10:  for  $i = 1$  to  $E$  do
11:     $R$  sends to  $u_i$  the coefficients  $q_j$  of  $Q(x)$  with degree more than  $d^*$ .
12:     $u_i$  computes  $S = P(u_i) + f(u_i) - \sum_{j=d^*+1}^d q_j u_i^j - g(u_i)$ .
13:  return  $S$ .
14:  end for
15: end procedure

```

Different from the original algorithm presented in [11], we use the random polynomials $f(x)$ of degree d instead of d^* . It is clear that adding $f(x)$ to $P(x)$ can hide all the coefficients of $P(x)$ which prevent the *reducer* from obtaining any information about the secret s . We also use another random polynomial $g(x)$ of degree d^* , where the constant term of $f(x)$ and $g(x)$ are identical. In the end of Algorithm 4, each cloud subtracts $g(u_i)$ from the result which will keep the secret s unchanged. To protect the secrets, for every degree reduction, the random polynomial $f(x), g(x)$ should be updated. In practical implementation, the dealer (with no *randomizer*) can secret share these polynomials to the clouds in advance or let clouds interact with the *randomizer*, thus supplies on-line these $f(x)$ and $g(x)$ pairs upon requests and the degree reduction process is performed with no involvement of the dealer during the execution.

In our proposed architecture, the read/write memory is separated from the read-only memory. This design is more convenient for degree reduction compared with the classic architecture. Compared with the whole memory space,

² The original algorithm is designed for bivariate polynomial, we modified it accordingly.

Algorithm 5. The SSS-Subleq plus degree reduction

```

1: procedure SSS-SUBLEQ-DR( $A, B, C$ )
2:   Decrease( $A||B||C||PC + 1, 3\ell, \ell$ )
3:    $R_1 \leftarrow \text{READ}(A)$ 
4:    $R_2 \leftarrow \text{READ}(B)$ 
5:    $R||Num = \text{SSS-SUB}(R_1, R_2)$ 
6:   DECREASE( $R||Num, *, \ell$ )
7:   WRITE( $B, R$ )
8:   JUMP( $Num \cdot C + (1 - Num) \cdot (PC + 1)$ )
9: end procedure

```

the read/write registers are very small, thus, the number of items for which we need to reduce the degree is relatively small. Assume that both the addresses and data items are secret shared using the polynomials of the same degree ℓ , plus degree reduction step, the Subleq can be implemented as in Algorithm 5. In step 6, we use $*$ instead of the exact degree parameter, as each secret shared bit of R has different polynomial degree.

4 Applications

In our model, assume that a client wants to outsource a program in clouds and the program is compiled into Subleq-based program. The address is encoded using unary representation and the data item is encoded using two's complement representation. The dealer picks random polynomials of degree ℓ to share every bit of the address and data. Then the dealer sends the secret shared program to E clouds. The integer E should be greater than the highest polynomial degree generated during Algorithm 5. Note that the participating clouds do not communicate with each other and are possibly not aware concerning the number and identity of the other participants. Also note that all the clouds (including *reducer* and *randomizer*) need not to be reliable.

Initial Stage. The PC of each cloud is initially set by the dealer. The values of the PC are the secret shares of the first address of the client's program. In case there is no *randomizer* in the system, the dealer can guarantee that each cloud has enough precomputed values of polynomials to be used for degree reductions.

Execution Stage. In this stage, the clouds have to perform the following works:

- Program Execution: Each cloud executes the secret shared program independently and does not communicate with other clouds.
- Degree Reduction: Each cloud performs Algorithm 4 to reduce the polynomial degree of the shares which increased during the Subleq procedure.

Memory Refresh. Although we decreased the polynomial degree of the shared secret before write, the operation *WRITE* does increase the polynomial degree by ℓ each time. Thus, the read/write part of memory needs to be refreshed at intervals (e.g., every ten *WRITE* operations). Note that this part of memory can

be relatively small compared with the whole memory, so it will not lead to too much bandwidth usage.

In Fig. 3, we give the outline of the program execution. The communication between the clouds and the dealer, and the communication between the clouds and the *reducer*(s) are all bidirectional. The dealer sends the secret shares of the client program and receives and reconstructs the program results executed by clouds. Moreover, we can use more than one *reducer* in order to check the integrity of the results and identify which *reducer* is malicious.

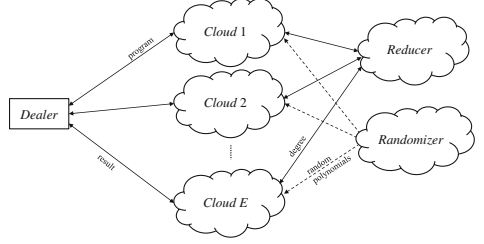


Fig. 3. The outline of Our RAM model

Storage and Bandwidth. The storage of each cloud consists of the secret shares of the program instructions and the data. Notice that secret share of one bit needs one or multi-word size storage which is denoted by $\omega(1)$.

Data Table. Each row of the data table consists of the index and data item, it totally requires $(n + k)(n + k + t)\omega(1)$ words storage. As we previously assumed that the size of read-only table is much bigger than that of the read/write table, i.e., $k \gg n$, the storage requires roughly $O(k^2)\omega(1)$ words.

Instruction Table. The cloud stores an instruction table of size m , and each instruction consists of five parts. This requires $O(m)$ blocks storage with each block requires $O(3m + 2n + 2k)\omega(1)$ words.

Degree Reduction Table. According to the corresponding description of Algorithm 4, if a *randomizer* (or several *randomizers*) are used to produce secret shares of random polynomials on-line, no tables are needed. Otherwise every cloud needs to store a certain amount of shares which are pre-computed and dispatched by the dealer. These values could be generated and managed by a special database. The size of this database is dependent on the execution length of the program, i.e., about $O(m\ell)\omega(1)$ words.

Bandwidth. For each Subseq, the clouds need to reduce the polynomial degrees of their data twice via communication with the *reducer* (and the *randomizer*). For each degree reduction from d to d^* , every cloud first obtains two shared evaluations from the *randomizer*, and then sends the *reducer* one word and receive $d - d^*$ coefficients from it, resulting in a total of approximately $O(k + m + t)\omega(1)$ words bandwidth used per cloud for one Subseq. In addition, the read/write memory needs to be refreshed at interval, it will result in $O(kt)\omega(1)$ words bandwidth usage. Therefore, in the worst case, the bandwidth of each cloud is $O(kt)\omega(1)$.

Security Analysis Sketch. We note that during the whole procedure of our model, all of the information are secret shared in E clouds and no original information will be leaked to the cloud itself. Besides this, our model has two characteristics:

Security Against Adversary Eavesdropping. For every LOAD operation, we had to compare the values stored in PC with all the indices in program table. It “touches” every position in the program table. Even through the adversary could eavesdrop on all the contents of PC , registers, etc., the adversary could not know which instruction in the table was executed. The same thing also happens in read/write operations. The characteristic is similar to the schemes that are based on fully homomorphic encryption, but here is information-theoretically secure.

Security Against Malicious Clouds. The malicious clouds include malicious participants and malicious *randomizer* and *reducer*. Informally, a malicious server can corrupt data in storage; and deviate from the prescribed protocol, particularly, not performing execution correctly.

For the participants: note that the program is outsourced to E clouds. Even if some of them output the wrong answers, the client can compare the results interpolated from different set of answers and obtain the correct result, or better off, use [30].

For the *reducer* and *randomizer*: every cloud may record the communication with the *randomizer* and *reducer* for audit, revealing possible malicious *reducers*. A possible strategy is to use several *reducers* simultaneously. After each cloud received the answers from the *reducers*, they could compare these results and notified the client/dealer whether the *reducers* were malicious or not. Similarly the actions of the *randomizer* can be monitored, say by forwarding the values sent by the *randomizer* to the *reducer*, requesting to the *reducers* to reveal all coefficients, and not use these values, requesting new values from the *randomizer*.

Unary vs. Binary. In our scheme, we use the unary representation for the instruction and data addresses. This type of representation is inappropriate if the clients program is very large because of its redundant bits. In a secret shared form, we have to use n words to represent these n bit which will lead to many operations over \mathbb{F}_p . As described in Sect. 2, we can use binary representation as a substitution. Compared with unary representation, binary representation can express exponentially more numbers with the same number of bits. However, using binary representation to perform secret string matching is more complicated and will require more degree reduction operations. In practical implementation, one can choose the representation based on the consideration of their memory and computation capacity.

5 Conclusions

We discussed a novel model for outsourcing arbitrary computations that provide confidentiality, integrity, and verifiability. Unlike the former RAM-based secure computation models, our scheme hides the client program and data all the time and manipulates the secrets directly. Therefore, no confidential information would be revealed. The setting is particularly interesting in the scope of big data that is stored in secret sharing fashion over the clouds, and there is a need to repeatedly compute functions over the data without reconstructing the data from the shares.

An important observation is that the dealer (and *reducer(s)*) may share common roots of all polynomials, unknown to the participating clouds, where addition and multiplications keep the roots unchanged. These unknown roots can serve as additional keys, the number of possible roots grows exponentially with the degree of the polynomials. Furthermore, implementation of interactive program is possible by reading and writing specific memory locations during the execution. Lastly, using several RISC instructions instead of OISC is possible to implement the program obviously. For every instruction execution, we can perform each instruction once and using secret string match technique to ensure the right execution.

References

1. Amazon Elastic Compute Cloud (EC2). <http://aws.amazon.com/ec2>
2. Afshar, A., Hu, Z., Mohassel, P., Rosulek, M.: How to efficiently evaluate RAM programs with malicious security, Cryptology ePrint Archive, Report 2014/759 (2014)
3. Becker, G.T., Regazzoni, F., Paar, C., Burleson, W.P.: Stealthy dopant-level hardware trojans. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 197–214. Springer, Heidelberg (2013)
4. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC 1988), NY, USA, pp. 1–10. ACM, New York (1988)
5. Boyle, E., Goldwasser, S., Tessaro, S.: Communication locality in secure multi-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 356–376. Springer, Heidelberg (2013)
6. Boyle, E., Chung, K.M., Pass, R.: Large-scale secure computation, Cryptology ePrint Archive, Report 2014/404 (2014)
7. Brenner, M., Wiebelitz, J., von Voigt, G., Smith, M.: Secret program execution in the cloud applying homomorphic encryption. In: Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST), pp. 114–119 (2011)
8. Brenner, M., Perl, H., Smith, M.: How practical is homomorphically encrypted program execution? An implementation and performance evaluation. In: IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 375–382 (2012)
9. Clash of the clouds. The Economist. http://www.economist.com/displaystory.cfm?story_id=14637206;2009
10. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 144–163. Springer, Heidelberg (2011)
11. Dolev, S., Garay, J., Gilboa, N., Kolesnikov, V.: Swarming secrets. In: 47th Annual Allerton Conference, pp. 1438–1445 (2009)
12. Dolev, S., Gilboa, N., Li, X.: Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation. In: Proceedings of the 3rd International Workshop on Security in Cloud Computing (SCC 2015), pp. 21–29. ACM, New York (2015)

13. Google Cloud Platform. <https://cloud.google.com/storage/>
14. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 169–178. ACM (2009)
15. Gentry, C.: A fully homomorphic encryption scheme, Ph.D. dissertation, Stanford University (2009)
16. Gentry, C., Halevi, S.: Implementing Gentry’s fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
17. Gentry, C., Goldman, K.A., Halevi, S., Jula, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: De Cristofaro, E., Wright, M. (eds.) PETS 2013. LNCS, vol. 7981, pp. 1–18. Springer, Heidelberg (2013)
18. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
19. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. *J. ACM* **43**, 431–473 (1996)
20. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: ACM Cloud Computing Security Workshop (CCSW) (2011)
21. HOMOMORPHIC ENCRYPTION. <http://sites.nyuad.nyu.edu/moma/projects.html>
22. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating efficient RAM-model secure computation. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP 2014), pp. 623–638. IEEE Computer Society, Washington, D.C. (2014)
23. Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013)
24. Mazonka, O., Kolodin, A.: A simple multi-processor computer based on subleq, arXiv preprint [arxiv:1106.2593](https://arxiv.org/abs/1106.2593) (2011). <http://da.vidr.cc/projects/subleq/>
25. Shamir, A.: How to share a secret. *Commun. ACM* **22**(11), 612–613 (1979)
26. Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013), NY, USA, pp. 247–258. ACM, New York (2013)
27. SUBLEQ. <http://mazonka.com/subleq/>
28. Tehranipoor, M., Koushanfar, F.: A survey of hardware trojan taxonomy and detection. *IEEE Des. Test Comput.* **27**(1), 10–25 (2010)
29. Wang, X., Huang, Y., Chan, T.-H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: The 21st ACM Conference on Computer and Communications Security (CCS), Scottsdale, Arizona, USA, November 2014
30. Welch, L., Berlekamp, E.R.: Error correction for algebraic block codes, US Patent, 4 633 470 (1983)
31. Zhuravlev, D., Samoilovych, I., Orlovskiy, R., Bondarenko, I., Lavrenyuk, Y.: Encrypted program execution. In: IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 817–822 (2014)

Algorithmic Aspects of Cloud Computing

First International Workshop, ALGO CLOUD 2015, Patras,
Greece, September 14-15, 2015. Revised Selected
Papers

Karydis, I.; Sioutas, S.; Triantafillou, P.; Tsoumakos, D.
(Eds.)

2016, XIV, 193 p. 49 illus. in color., Softcover

ISBN: 978-3-319-29918-1