A Flexible Scheme of Garbled Random Access Stored-program Machine

Fu-Te SHANGa*, Chuan-Rong ZHANG
School of Information and Navigation, Xi'an City, Shaanxi Province, China
ashangfute@163.com
*Corresponding author

Keywords: Garbled Circuit, One Way Function, Random Access Stored-program Machine (RASP), Private Function Evaluation.

Abstract. Recent work on garbled RAM has resulted in built systems that garble RAM program in secure multiparty computation protocol directly, avoiding the inefficient process of first converting the program into a circuit. But most of these schemes need to preload the hardwired circuits, and the amount of communication is huge.

This paper uses the garbled RAM scheme from one way function (Garg, Lu, Ostrovsky, Scafuro) to construct garbled random access stored-program machine, which is capable of modifying program at the run-time by introducing garbled code memory and control circuit. Through comparing garbled circuit with garbled RAM approach, a reduced amount of data communication can be achieved by applying our algorithm, which is proved by theoretical analysis. As an independent advantage, our scheme with full security seamlessly supports private function evaluation.

Introduction

Clients may need to store some dataset on the cloud, and then may want the cloud to execute program on the dataset. If they expect to expose nothing about the dataset and the computation to the cloud, they need to process the remote data through the approach of secure computation.

Early secure computation schemes need to first compile the program to circuits, including Yao’s approach[1] and the approach based on secret sharing. The recent fully homomorphic encryption[2] method also need to first compile programs to circuits to realize the computation on encrypted data. But if we convert the programs into circuits, the size of the programs will grow with the size of the input. Of course, except representing programs as circuits, there are other methods, such as randomizing polynomials and encoding functions. But many algorithms can be more compact and natural to present as RAM program. For example, in the scenario of large encrypted database query, the converted circuits should be able to compute all the possible execution paths of the original program[3]. Consequently, the circuit size will grow exponentially with the increase of database size. Using fully homomorphic encryption can reduce the size of input. However, it’s still necessary to compute on the entire encrypted database for the server. Motivated by the considerations above, garbled random access machine[4-7] (GRAM) was developed.

The scheme of GRAM can garble RAM program directly, and keep the same security property as Yao’s garbled circuits[8]. These schemes can solve the efficiency problem of garbled circuits, but there exist two flaws as follows.

1. Almost all known GRAM schemes preload the hardwired circuits to execute the program, which is difficult to load program dynamically at run-time. But many programs cannot decide the length of loop or recursion depth at the compile phase.

2. In the GRAM schemes, the loop and recursion structure need to be unfolded to a sequence structure at the compile phase, so the length of the garbled program is related to the actual running time, not to the length of plain program length, which may lead to more communication resources and is hard to fit the need of cloud computation.
This paper is based on the GRAM construction proposed by[4] and introduces the concept of garbled code random access memory and the control circuit to construct the garbled random access stored-program machine (GRASP). In the proposed GRASP scheme, garbled code is the sequence of CPU instructions stored in the garbled memory, the size of which is only related to the length of plain program. The instructions can be randomly accessed and executed by the control circuits. Consequently, the above problems can be solved efficiently.

Background
In this section, we fix the notation of garbled circuit and garbled random access machine, which is the cited from the construction by Garg et al.

Garbled Circuit
The garbled circuit scheme[9] is a tuple of PPT algorithms (GCircuit, Eval), where GCircuit is the circuit garbling procedure, Eval is the circuit evaluation procedure. Every wire $w$ of the circuit has two labels $\text{label}_w^0$, $\text{label}_w^1$. Every output wire can assign any label as well to blind the output through generic transformation. We also have a Test algorithm to test the form of given labels.

1) $(\tilde{C}, \{(w, b, \text{lab}_w^b)\}_{w \in \text{inp}(C), b \in \{0,1\}}, \{(w, b, \text{lab}_w^b)\}_{w \in \text{out}(C), b \in \{0,1\}}) :$ given a security parameter $\kappa$, a circuit $C$, and the label set $\text{lab}_w^b$ of all output wires $w \in \text{out}(C)$, where $b \in \{0,1\}$, $\text{out}(C)$ represent the set of all output wires, this procedure output the garbled circuit $\tilde{C}$ and the labels set $\text{lab}_w^b$ of all input wires $w \in \text{inp}(C)$.

2) $0/1 \leftarrow \text{Test}(\text{lab}) :$ efficiently test if the labels is meant for a garbled circuit.

3) $y = \text{Eval}(\tilde{C}, \{(w, \text{lab}_w^b)\}_{w \in \text{inp}(C)}) :$ given the garbled circuit $\tilde{C}$ and the set of input labels $\{(w, \text{lab}_w^b)\}_{w \in \text{inp}(C)}$, output the result $y$.

Garbled Random Access Machine
The scheme by Garg et al. allows multi-program to execute sequentially after garbling the data once and the garbled memory modification persists during the execution. Following definition only satisfies the condition of garbling single program and assumes the security with unprotected memory access (UMA).

An UMA-secure single program garbled RAM scheme consists of four PPT algorithm (GData, GProg, GInput, GEval):

1. $(\tilde{D}, s) \leftarrow \text{GData}(1^\kappa, D) :$ given a security parameter $1^\kappa$ and memory $D \in \{0,1\}^m$ as the input, GData outputs the garbled memory $\tilde{D}$.

2. $(\tilde{P}, s^m) \leftarrow \text{GProg}(1^\kappa, 1^{\text{log}m}, 1^l, P) :$ given a RAM program $P$ and the memory size $m$ as input, GProg outputs the garbled program $\tilde{P}$ and an input-garbling-key $s^m$.

3. $\tilde{x} \leftarrow \text{GInput}(1^\kappa, x, s^m, s) :$ given a program input $x \in \{0,1\}^l$, an input-garbling-key $s^m$ and a root key $s$ as input, GInput outputs the garbled input $\tilde{x}$.

4. $y = \text{GEval}^0(\tilde{P}, \tilde{x}) :$ given a garbled program $\tilde{P}$, a garbled input $\tilde{x}$ and the garbled memory $\tilde{D}$, GEval outputs the result $y$.

The Construction
In this section, we describe our definition of basic random access stored-program machine (RASP) and the garbled RASP construction.
Basic Random Access Stored-program Machine

For the convenience of description, Based on the reduced instruction set computer (RISC) with word size $W$, the instructions are summarized in Table 1 below. Every instruction consists of an opercode and three operands, takes one cycle of the machine to execute and needs to random access the memory up to one time. Assume that there is a code of running time $T$ stored in a code memory $P \in \{\text{instr}\}^n$ of size $n$, and a data memory $D \in [0..2^W - 1]^m$, where $4n + m < 2^W$. $P^D(x)$ represents one execution of code $P$ on data $D$ with initial input $x \in \{0,1\}^t$. During the execution, RASP could access arbitrary location of data memory $D$ and code memory $P$. The code $P$ can represented as a sequence of $T$ small CPU-Step Circuit. One CPU step can be denoted by:

$$C_{\text{CPU}}(\text{state}, d^{\text{read}}, i^{\text{read}}) = (\text{state}', L_D, L_P, d^{\text{write}}, i^{\text{write}})$$

Table 1. Instruction Set in This Model.

<table>
<thead>
<tr>
<th>Calculation Instructions</th>
<th>AND</th>
<th>NOT</th>
<th>MUL</th>
<th>SHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>ADD</td>
<td>DIV</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>SUB</td>
<td>NOP</td>
<td>CMP</td>
<td></td>
</tr>
<tr>
<td>Memory Instructions</td>
<td>MOV</td>
<td>STORE</td>
<td>LOAD</td>
<td></td>
</tr>
<tr>
<td>Control Instructions</td>
<td>JMP</td>
<td>JZ</td>
<td>JNZ</td>
<td>HALT</td>
</tr>
</tbody>
</table>

The circuit takes current CPU state $\text{state}$, a data block $d^{\text{read}}$ and an instruction $i^{\text{read}}$ as input, where the data and instruction is retrieved according by previous CPU step requested location. CPU step circuit takes updated state $\text{state}'$, next data location $L_D \in [m]$ and instruction location $L_P \in [n]$ to read, and the data block $d^{\text{write}}$ and instruction $i^{\text{write}}$ to write to the previous location as input. The sequence of location and read/write values is called access pattern, which consists of data access pattern $\text{MemAccess}_D = \{(L'_D, d^{\text{read}}, d^{\text{write}}), \text{state}: \tau = 1, \ldots, t\}$ and code access pattern $\text{MemAccess}_P = \{(L'_P, i^{\text{read}}, i^{\text{write}}), \text{state}: \tau = 1, \ldots, t\}$. Without loss of generality, we assume the location to read and write is the same, which is sufficient to access arbitrary location through a dummy read and write step.

The RASP computation can be presented as CPU step circuits. Assuming the initial state $\text{state}_0$, initial read location $L^0_D = 0$, $L^0_P = 0$. For each $\tau \in \{1, \ldots, T - 1\}$ proceed as follows:

1) Instruction Fetch: fetch the instruction according to the location $i^{\text{read}}: P[L'^{\tau}_{\text{read}}]$.
2) Data Read: read the data according to the location on the instructions $d^{\text{read}}: D[L'^{\tau}_{\text{read}}]$.
3) Execute: execute the instruction, update $\text{state}', L'_D, L'_P, d^{\text{write}}, i^{\text{write}}$.
4) Write Back: write data and code to the memory $P[L'_{\text{write}}] = i^{\text{write}}, D[L'_D] = d^{\text{write}}$.

Finally, if $\tau = T$, then we set $\text{state}_T$ to be the output of the program $P$.

Garbled Random Access Stored-program Machine

Garbled RASP also has four steps: data and code garbling, garbled circuit generation, initialization and circuit evaluation as follows.

$\text{GData}$ was adopted to garble data and code. We need to convert the code $P$ to binary form $\hat{P} \in \{0,1\}^{4 \cdot 4W}$ first and compute $(\hat{M}, s) \leftarrow \text{GData}(\text{state}_T, \hat{P} \parallel D)$, where $M = \hat{P} \parallel D$. $\hat{M}$ represents a garbled memory including garbled code segment $\hat{P}$ and garbled data segment $\hat{D}$, whose starting locations are $L^0_D, L^0_P$. $\text{state}_0$ consists of the registers $\text{regs, flag}$ and locations $L^0_D, L^0_P$. 
\((\tilde{C}, s^{in}) \leftarrow \text{GPprog}(1^\kappa, 1^{logm}, \lambda, C)\) is used to garble the following sequence of \(T\) CPU step circuits  
\(C^{\text{CPU}}_{\text{state}_x, d^{\text{read}_r}, i^{\text{read}_r}}\) to get the final garbled RASP \(\tilde{C}\). For each \(\tau \in \{1, \ldots, T - 1\}\) proceed as follows:

1) Instruction Parse: \((op, src, dst, L_D^{\tau + 1}, imm)\leftarrow \text{Parse}(1^W, i^{\text{read}_r}, d^{\text{read}_r}, \text{state}_x)\) to get the information of instructions.

2) Calculation Execute: if \(op = \text{ADD}\), then: \(s^{src} = s^{dst} + s^{\text{dest}}\); else if \(op = \text{SUB}\), then: \(s^{src} = s^{dst} - s^{\text{dest}}\); else if \(\ldots\) (more other calculation instructions)

3) Instruction Location Update: if \(op\) is a calculation instruction or memory instruction, then \(L_p^{\tau + 1} = L_p^{\tau} + 4\); else if \(op = \text{JMP}\), then \(L_p^{\tau + 1} = L_p^{\tau} + 4 + 4 \times \text{imm}\); else if \(op = \text{HALT}\), then \(L_p^{\tau + 1} = L_p^{\tau}\); else if \(\ldots\) (more conditions need to update the instruction location)

4) Memory Access: it’s necessary for all types of instructions to read the memory \(i^{\text{read}_r, \tau + 1} = P[L_p^{\tau + 1}]\), \(d^{\text{read}_r, \tau + 1} = D[L_m^{\tau + 1}]\) and write to the memory \(P[L_p^{\tau + 1}] = src\), \(D[L_m^{\tau + 1}] = src\).

The initialization of garbled RASP needs \(\tilde{x} \leftarrow \text{Glnput}(1^\kappa, x, s^{in}, s)\). It takes the security parameter \(\kappa = W\), the initial state \(x = (\text{state}_0, d^{\text{read}_0} = \perp, i^{\text{read}_0} = \text{nop})\), a garbled circuit \(\tilde{C}\) and an input-garbling-key \(s^{in}\) and root key \(s\) of the garbled memory \(\tilde{M}\), and outputs \(\tilde{x}\).

During the evaluation phase, we use \(y = \text{GEval}^\kappa(\tilde{C}, \tilde{x})\) to get the result \(y\).

Security Proof and Performance Analysis

Security Proof

Garg et al. had proven that given one-way functions, an UMA secure garbled RAM scheme exists. Consequently, for any efficient attacker \(A\), there exists a PPT simulator \(\text{Sim}\) for any code \(P\), initial memory \(D \in \{0, 1\}^m\) and input \(x\), access pattern \(\text{MemAccess}\):

\[(\tilde{D}, \tilde{P}, \tilde{x}) \approx \text{Sim}(1^\kappa, 1^m, \lambda, y, D, \text{MemAccess})\]

where \((\tilde{D}, s) = \text{GData}(1^\kappa, D), (\tilde{P}, s^{in}) = \text{GPprog}(1^\kappa, 1^{logm}, \lambda, P)\), \(\tilde{x} = \text{Glnput}(1^\kappa, x, s^{in}, s), y = \text{P}^\kappa(x)\).

**Theorem 1**

Given an UMA secure garbled RAM scheme, then for any initial data contents \(D\) and any code \(P\) of running time \(T\), there exists an UMA secure garbled RASP scheme.

**Proof Sketch**

The correctness of this scheme follows by combining the fact that the correctness of garbled RAM scheme was proven and that our code is first encrypted in the garbled memory then executed by the control circuits correctly. Consequently, our code dynamically executed follows the original program logic.

For any efficient attacker \(A\), the simulator \(\text{Sim}\) in our scheme is nearly identical to the UMA secure garbled RAM scheme. This is because the joint distribution of the garbled circuit \(\tilde{C}\) constructed from the sequence of \(T\) CPU step circuits, the garbled memory \(\tilde{M}\) and the garbled input is computational identical to the distribution in the garbled RAM scheme. Consequently, there exists the simulator that takes an initial data contents \(D\), a protocol output \(y\), an access pattern \(\text{MemAccess}\) and a secure parameter \(\kappa\), a running time \(T\) of the code as input, which makes this scheme have the UMA-security.

We can obtain a garbled RASP scheme with the full security by combining the Oblivious RAM (ORAM) and an UMA secure garbled RASP scheme.
Theorem 2

Assuming an UMA secure garbled RASP scheme and ORAM (both of which could be efficiently constructed from OWFs) exist, there exists a garbled RASP scheme with the full security.

The proof proceeds similarly to the garbled RAM scheme with the full security case and we skip the details here. Additionally, the garbled RASP scheme with the full security support other security property of private function evaluation, in which one party’s private input is a program, the other’s private input is the data. The default definitions of garbled RAM do not include program privacy, and the garbled program \( \tilde{C} \) may reveal information about the code of the actual program \( C \) to the server. The code is encrypted in the garbled memory in our scheme. The only information leakage is about the running time of the program. A malicious attacker may utilize this to get some information about the private data. But in our construction, when the program reaches a \texttt{HALT} instruction, \( L_p \) will maintain its value, rather than exit. Therefore, the running time is independent to the data, avoiding the above problem.

Performance Analysis

Taking advantage of our garbled RASP scheme, we can construct many programs which can be hard to realize in a garbled RAM scheme. By allocating a function calling stack in garbled memory it can be constructed easily to build recursion structure and other modern program language features.

To show the time and space performance, we compare the GC scheme, the GRAM scheme and the GRASP scheme. Assuming the word size is \( W \), the security parameter is \( \kappa \), the program \( P \in \{\text{instr}\}^n \) with running time \( T \) and plain code size \( n \), the data \( D \in \{0,1\}^m \). We show the performance comparison in Table 2.

<table>
<thead>
<tr>
<th>Garble Scheme</th>
<th>Memory Garbling Time</th>
<th>Circuit Garbling Time</th>
<th>Garbled Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>( O(m) )</td>
<td>( O(T \cdot m) )</td>
<td>( O(T \cdot m) )</td>
</tr>
<tr>
<td>GRAM</td>
<td>( m \cdot \text{polylog}(m) )</td>
<td>( T \cdot \text{poly}(\kappa) )</td>
<td>( O(T \log m) )</td>
</tr>
<tr>
<td></td>
<td>( \text{poly}(\kappa) )</td>
<td>( \text{polylog}(m) )</td>
<td></td>
</tr>
<tr>
<td>GRASP</td>
<td>( T \cdot \text{poly}(W) )</td>
<td>( \text{poly}(W) )</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>

As we can see from the table, the GRAM scheme is better than the GC scheme in all aspects. The time complexity in the GRASP scheme is worse than the GRAM scheme, but the garbled code size in the GRASP scheme is much better than that in the GRAM scheme. If we need to execute a new code during the protocol, the garbled code generator needs to send it to the other party, and then the communication amount will be very different.

Summary

In this work, we designed a garbled RASP scheme to solve the garbled circuit problem in secure multiparty computation by introducing the garbled code memory and the CPU control circuit to the garbled RAM scheme. By supporting the program control flow, we allow programmers not just to be able to modify program at the run-time, but also to reduce the communication amounts. Through ORAM, we can obtain the full security and private function evaluation at the same time.

There are still many interesting, unexplored optimizations that would further improve the security and the efficiency of our approach: In our garbled RASP scheme, data and code was stored in the same memory, which has the risk of memory overflow attack. If an attacker writes a data area over its bound to some code area, he could be able to hijack the control flow of the program. To alleviate this vulnerability, we can store the code and data to the separated memories to make it difficult to attack
the code. But we may need more circuits to access different memories. Through rearranging the calculation procedure and designing a two phase pipeline to optimize the time complexity up to one half. There are many other approaches to alleviating the attack in modern computer system, which can be associated with our scheme.

**Acknowledgement**

This research was financially supported by the National Science Foundation.

**References**


