CS 294 – Secure Computation Mar 1-3, 2016

Lecture 13: Secure RAM Computation and Fairness

Instructor: Sanjam Garg *Scribe: Arjun Baokar*

1 Introduction to Secure RAM Computation

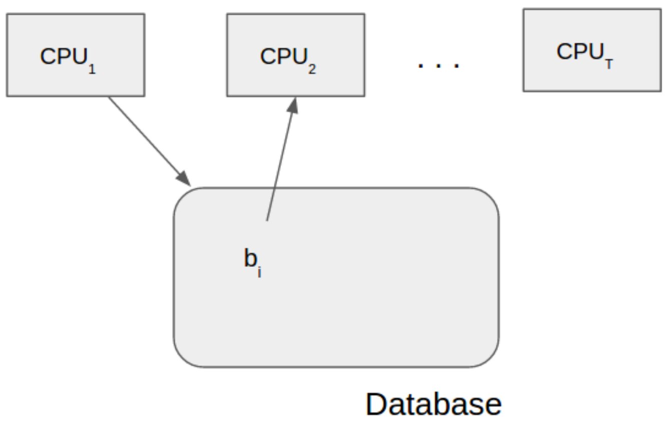
The goal of secure RAM Computation is to run a program P which accesses a shared database D without revealing its access patterns. We can leverage Oblivious RAM to do this, since ORAM allows us to compile P into $\widetilde{P}$ and D into $\widetilde{D}$ and have $\widetilde{P}$ accesses to $\widetilde{D}$ not reveal P's accesses to D. Given, this we can construct a solution that doesn't hide access patterns, and let ORAM handle hiding access patterns.

We want to avoid direct circuit conversion of the RAM program, since this greatly increases complexity. If a RAM program runs in time $O(T)$, then its circuit runs in $O(T^3)$. If its database accesses are $O(D)$, then the circuit's accesses scale with T and D, because the circuit needs to take D as input. Our goal is to do this in polylog time.

An important thing to keep in mind is that we don't know what has to be read beforehand when working with a RAM program. If we did, then the circuit complexity wouldn't be bad, and we could just use that. The motivating example for the notes can be a binary search on D shared between two parties, where we want to find the solution in polylog time without revealing the query.

2 RAM Program

A RAM program can be looked at as a series of CPU steps, each which accesses the database (either a read or write). Each CPU step changes the state for the subsequent step.

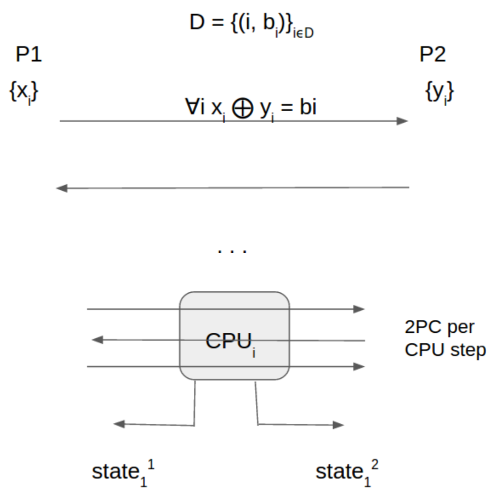


3 Secret-Shared Database Construction

One way we can do this is by secret sharing each bit of the database D. A simple way to do this by defining the database as such:

Definition 1 $D = \{(i, b_i)\}_{i \in |D|}$

Both parties receive $\{x_i\}$ and $\{y_i\}$ respectively, where $\forall i, x_i \oplus y_i = b_i$.



This construction has high round complexity. From an information theoretic perspective, to have small round complexity, one person must have all data. This is because of the reason mentioned above: we don't know what value to read (we don't know what's in memory to be able to read the next value at a specific CPU step). We'll discuss how to improve this in the next section.

4 Encrypted Database Construction

To give one person the data, we can have one person own the the encrypted version of the database. The other person can have the key to the database. Formally:

Definition 2 $D_i = Enc_K(b_i)$ *given to one user as* y_i*, and* K *given as* x_i*.*

They can still only access the database together. The same protocol as the secret-sharing case applies, just replacing x_i and y_i as mentioned in the definition. However, this doesn't quite work, since we don't really have security here (y_i's owner can figure out D_i since K is sent over by the other person).

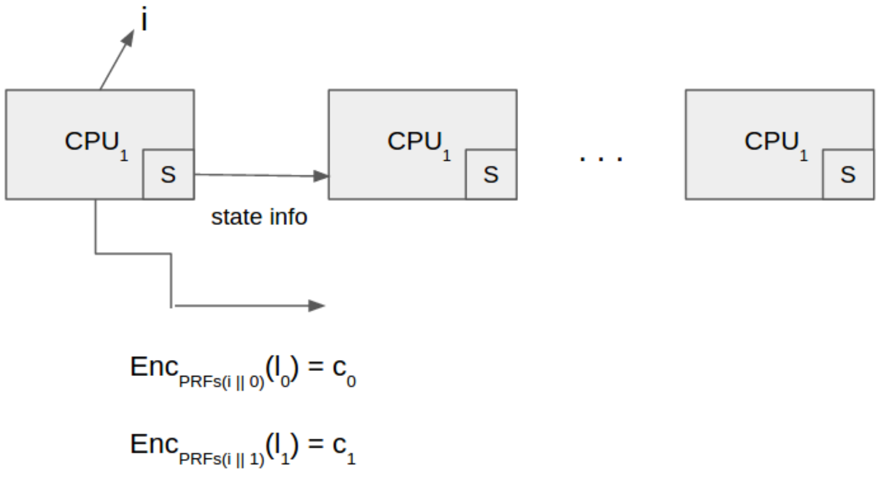
Goal: We want to implicitly decrypt without the encrypted database's owner finding out about D_i. We can garble each CPU step separately:

Definition 3 *$Garble(CPU_i) = \widetilde{CPU_i}$*

$\widetilde{CPU_i}$ will output labels for $\widetilde{CPU_{i+1}}$. We create a κ-bit key sampled randomly.

Definition 4 *$S = \{0,1\}^\kappa$ randomly sampled.*

In the database, $\forall i$, we define $e_i = PRF_S(i||b_i)$, and store e_is in the database. This PRF is hard-coded into each of the $\widetilde{CPU_i}$s, so each garbled circuit can use it.



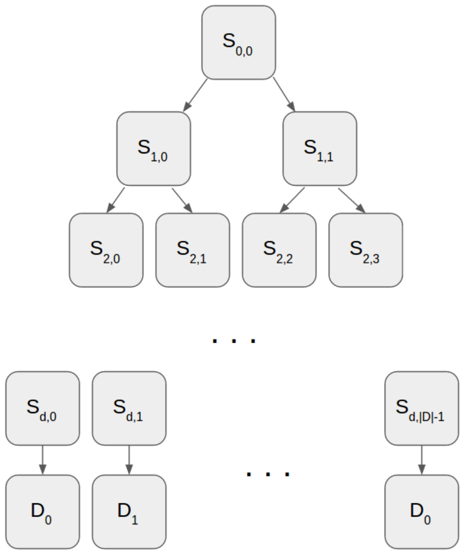
Focusing on just $\widetilde{CPU_1}$ and $\widetilde{CPU_2}$, we can how the labels for the circuits are transmitted forward. State labels are known, so $\widetilde{CPU_1}$ can easily provide those to $\widetilde{CPU_2}$. However, l_0 and l_1 need to be emitted based on what b_i is (emit l_{b_i}). To do this, $\widetilde{CPU_1}$ can emit $Enc_{PRF_S(i||0)}(l_0)$ and $Enc_{PRF_S(i||1)}(l_1)$ for l_0 and l_1 respectively.

We can then simulate these garbled circuits to evaluate security. Since garbled circuits require an output value for simulation, and only the last CPU step has an output, we can essentially replace each middle garbled circuit with "garbage" until the last circuit emits. However, a problem is that in garbled circuit simulation, only one label can be known (the one corresponding to l_{b_i}). The other should be indistinguishable from a random string. While this is trivially true for $\widetilde{CPU_1}$, for subsequent steps (say $\widetilde{CPU_2}$), it is not. $PRF_S(i||0)$ and $PRF_S(i||1)$ are known to all circuits, since S is harcoded into each circuit, as mentioned before. We need to expunge one label to be able to simulate.

Note: This is actually a circular argument at the moment. The garbled circuit depends on PRF_S to be random, and PRF_S requires the GC to be secure, since S is hardcoded into the circuit.

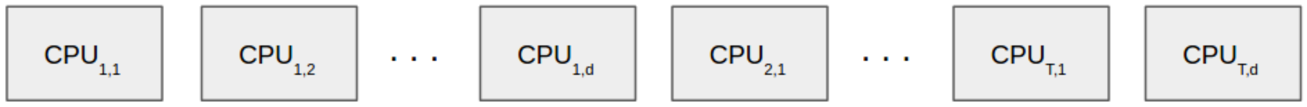
4.1 Fixed Construction

To fix the problem that everyone knows S, we can have more than one key S. Let's make a tree of keys, where keys are denoted $S_{i,j}$ with i as depth and j as their index within the depth.



In this tree, each child key is encrypted with its parent key (eg. $S_{1,0}$ is encrypted with $S_{0,0}$). If you know a key, you can decrypt its immediate children. The leaf nodes, $S_{d,i}$ for $i \in 0, |D|-1$, encrypt the bits of the database D. With the root key, you can follow the path down to any leaf and read its corresponding $b_i \in D$.

Now, for every $\widetilde{CPU}_i$ we had before, we create $lg(|D|)$s (we have logarithmically more CPU steps). Each of these logarithmically many CPU steps (shown below) will recreate this path through the key tree.



Each circuit will also replace the key it used with a fresh key (a new randomly sampled key) after it uses it once, and recreate the corresponding ciphertexts. This means that all siblings along the path (in addition to the path) will need to be recreated. Fixing these siblings is logarithmic time. This fixing step maintains the invariant that after each step, the subsequent step has a fresh copy of D as if it was never read.

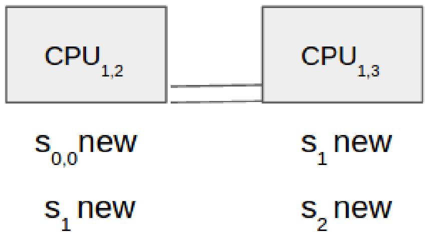
4.1.1 More Details on Key Generation

The key tree requires that each node encrypt its children under itself. There are two definitions that will help us explain the key tree usage.

Definition 5 *$e_{i,j} = \{PRF_{S_{i,j}}(k||(S_{i+1,2j}||S_{i+1,2j+1})_k)\}_{k \in 2\kappa}$, where k is the current key (of length κ) and $e_{i,j}$ is the encryption of the children under k.*

Definition 6 *$CPU_{i,j}$ ($i \in \{1...T\}, j \in \{1...d\}$) is a CPU step from the logarithmically expanded CPU step garbled circuits we mentioned before. $CPU_{i,j}$ sends labels to $CPU_{i,j+1}$, unless $j = T$, in which case, it sends labels to $CPU_{i+1,0}$.*

With this in mind, we can pre-generate some keys that we know we will need. Every $CPU_{t,1}$ and $CPU_{t,2}$, $t \in 1...T$ requires a fresh $S_{0,0}$ key, so we can hardcode these into those CPU step circuits beforehand. Similarly, each circuit has two fresh keys encoded, one to encrypt the labels it must pass on to the next circuit and one for which is the fresh encryption key used by its predecessor to encrypt the labels given to it. Now, to understand the process, we'll focus on 2 CPU steps: $CPU_{1,2}$ and $CPU_{1,3}$.



For simplicity, let's assume $S_{1,0}$ encrypts the correct label to send. Then, $CPU_{1,2}$ outputs $c = \{PRF_{S_{0,0}^{new}}(k||(S_{1,0}^{new}||S_{1,1}))_k\}_{k \in 2\kappa}$. For the case where $S_{1,1}$ is the correct key, simply swap $S_{1,0}$ and $S_{1,1}$. Note that in the figure above, $CPU_{1,2}$ has the S_1^{new} key. This key will become $S_{1,0}^{new}$ or $S_{1,1}^{new}$, depending on which one to send. Similarly, $CPU_{1,3}$ has S_2^{new} to use to encrypt the labels it sends. It also has S_1^{new} to decrypt what $CPU_{1,2}$ sends it.

5 Fairness

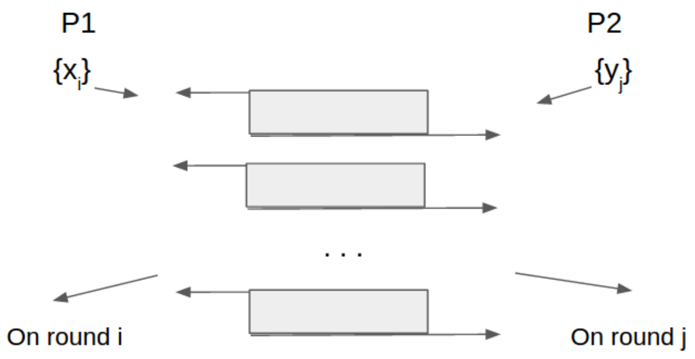
Definition 7 *Fairness dictates that anytime the bad guy gets an output from an MPC protocol, the good guy must also get the same output. These are its only guarantees.*

Let us assume a two-party setup with parties $P1$ and $P2$. $P1$ has inputs x_i and $P2$ has inputs y_j where i, j have a small domain. For the example in this section, let's assume $i, j \in \{1...6\}$. As a side note, it might help to think of this as the Yao's Millionaire Problem. Let us define our function we are trying to compute on the inputs:

Definition 8 *$f(x_i, y_j) = 1$ if $i > j$, 0 if $i \leq j$.*

The outputs from $f(x_i, y_j)$ forms a lower triangular matrix: $\begin{pmatrix} \begin{array}{c|cccccc} & \underline{y_1} & \underline{y_2} & \underline{y_3} & \underline{y_4} & \underline{y_5} & \underline{y_6} \\ x_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_3 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_4 & 1 & 1 & 1 & 0 & 0 & 0 \\ x_5 & 1 & 1 & 1 & 1 & 0 & 0 \\ x_6 & 1 & 1 & 1 & 1 & 1 & 0 \end{array} \end{pmatrix}$

The key idea is that when the a party gets their output depends on their input, so the attacker doesn't know when a good guy gets the output. However, if the attacker gets an output and then aborts, the good guy will be able to figure out what the output was. We assume a semi-honest setting with the ability for any party to abort (so a little stronger adversary than semi-honest).



We use a series of 2PC protocols, and assume that if the attacker aborts, they got the answer. In our case, since the domain of the inputs has a size of 6, we use 6 2PC protocols. Furthermore, $f(x_i, y_j)$ will only emit the answer x_i on the ith round and the answer for y_j on the jth round. Thus, each party can get one of 4 results from each round: $\{1, 0, \perp, ABORT\}$. We will prove this works by examining each of the cases, assuming $P1$ is the adversary and $P2$ is honest. An analogous argument can prove the other case.

- **Case 1**: No aborts. This is trivially fair.
- **Case 2**: Abort on round k, where $k < i$. This means that $P1$ didn't get the answer (she only has gotten $\perp$ up to the ABORT), so it doesn't matter what we do; it's still fair.
- **Case 3**: Abort on round k, where $k \geq i$. This means that $P1$ had to have emitted $f(x_k, y_j) = 0$, which is also what $f(x_i, y_j)$ must be, because it's a lower triangular matrix.

It is important to remember are that k is based on the first ABORT $P2$ gets.