Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 13: Garbled RAM

Scribe: Justin Zhang

May 1, 2024

Today, we discuss garbled RAM, a paradigm for random access memory (RAM) secure multiparty computation (MPC). In particular, we initially look at garbled circuits implemented by Andrew Yao[**?**], and then we discuss its extension into the garbling of RAM programs using ideas from recent sublinear computation construction [**?**][**?**] without increasing round complexity.

1 Secure Multiparty Computation

While classical cryptography (encryption, authentication, etc.) is about securing communication, modern cryptography (ZKP, MPC, etc.) is about securing computation. Informally, the premise for MPC is a group of mutually distrusting parties, each with a private input. Their objective is to learn the result of an agreed-upon computation, such as an election, auction, etc. with the security guarantees being privacy (everyone only learns the output), input independence, and output consistency. MPC can be more formally defined using a simulator and corrupted parties. For a reference, see [**?**].

2 Yao's Garbled Circuits

Yao's garbled circuits allow for circuit-based multi-party computation. For simplicity, we describe a two-party secure computation between Alice and Bob, who compute the result of a function $f(x, y)$ representable by a single gate taking in an input bit x from Alice and an input bit y from Bob. The truth table of this function is

Alice's Input x	Bob's Input y	$f(x, y)$
0	0	$f(0, 0)$
0	1	$f(0, 1)$
1	0	$f(1, 0)$
1	1	$f(1, 1)$

. In Yao's protocol, Alice acts as the "garbler," and Bob acts as the "evaluator." Suppose that they agree on some encryption scheme (G, E, D). Alice writes out a garbled, encrypted truth table for Bob as follows:

1. For input x, choose random crypto keys A_0, A_1. Symmetrically, for input y, choose random crypto keys B_0, B_1.
2. Encrypt each output with the corresponding keys. That is, for input (x, y), Allice encrypts $E_{A_x}(E_{B_y}(f(x, y)))$.
3. Randomly permute the outputs and send to Bob.

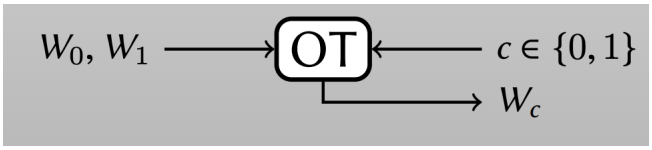

Figure 1: 1-out-of-2-OT [**?**]

Intuitively, if Bob can learn the correct keys A_x, B_y without learning their shared input (x, y), then Bob can only correctly decrypt the desired function output. To do exactly this, we use an **oblivious transfer** (OT) protocol.

In a 1-out-of-2-OT, Alice has two items a_0, a_1, where Bob wants to learn $a_b, b \in \{0, 1\}$.

In the protocol, Alice inputs β_0, β_1 and Bob inputs γ. At the end of the protocol, Alice learns nothing of Bob's bit γ, and Bob learns β_γ and nothing else. For a detailed specification of OT protocols, see [**?**].

One last technicality in this garbled circuit protocol is how will Bob know when he has decrypted the correct cipher? An easy solution to this is simply to prepend or append a string of 0s and have Bob check this. If we assume that we are using a secure encryption, then we have false positives with a negligible probability. Formally,

Modified Encryption

- $\mathsf{Gen}(1^\lambda) : k \leftarrow \{0,1\}^n$
- $\mathsf{Enc}_\mathsf{k}(m) : r \leftarrow \{0,1\}^n$. Output $(r, (0^n || m) \oplus \mathsf{PRK}_\mathsf{k}(r)$
- $\mathsf{Dec}_\mathsf{k}(c_1, c_2)$: Compute $m_0 || m_1 \leftarrow c_2 \oplus \mathsf{PRF}_k(c_1)$. If $m_0 = 0^n$, output m_1. Else output $\perp$.

Thus, the protocol is complete, where Alice will send a garbled truth table, Bob will request the correct keys via a 1-out-of-two OT, and Bob will successfully decode the correct output except with negligible probability.

2.1 Extending to Many Gate Circuits

The extension of this single gate protocol to a multi-gate protocol uses the same idea to propagate gate keys. Instead of simply returning the function output plaintext, every wire has two corresponding keys, where the communicated result from a gate is the next wire key.

For instance consider the 2-fan-in AND gate with input wire keys A_0, A_1 and B_0, B_1 and output wire keys C_0, C_1. Then Alice's communicated table (before permuting) is illustrated by figure **??**.

Thus, for arbitrary circuits, Bob will incrementally learn the wire keys corresponding only to the circuit computation on their input, until he learns the final function output.

In conclusion, 2PC can be performed using Yao's garbled circuit, which can be done in 2 rounds by having Alice send all the truth tables at once.

3 Garbled RAM

Next, we discuss the problem of garbling RAM programs. Many of the practical programs and functions (eg graph traversal) we care about rely on RAM. Since the accesses and writes of these programs are often dynamical and unpredictable in nature, they are not naively transformable into circuits without a large overhead. A naive translation of memory accesses to circuits is to implement a *linear array circuit*, which takes in an access index i and the memory array A, and outputs $A[i]$ via a linear scan.

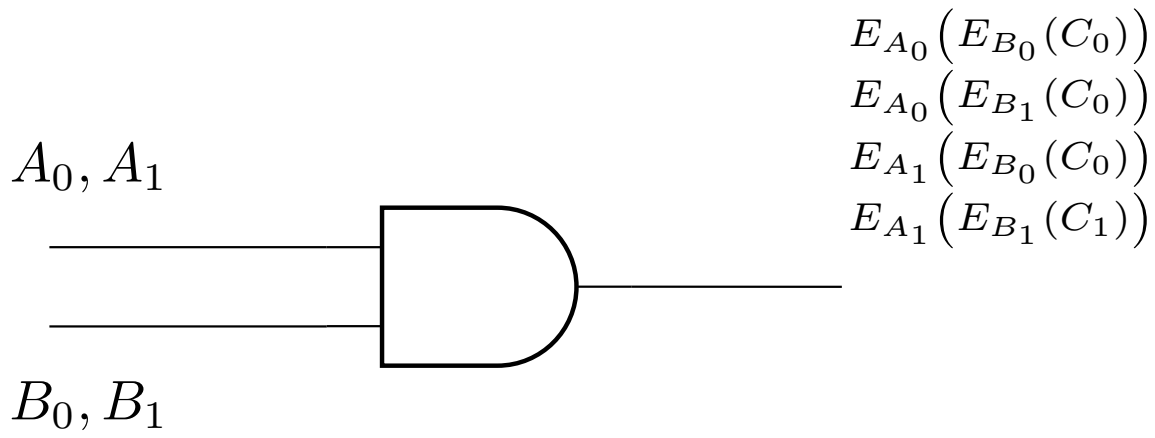

Figure 2: Key Propagation in a garbled AND Gate

Thus, generically, if T is the RAM program's run time and n is the size of the memory array, then naive circuit transformation with linear access circuits gives us $O(Tn)$ run time. A main challenge of garbled RAM is doing better than linear blow-up by garbling these dynamical accesses with sublinear cost, while keeping the two-round complexity of the original garbled circuit.

3.1 RAM Model 2PC: ORAM + 2PC

We start with a conceptually simpler protocol if we disregard the number of communication rounds.

Suppose that we have ORAM and 2PC protocols. Suppose that we have some non-oblivious RAM program between Alice and Bob. Then, we can apply an ORAM protocol to make it oblivious.

Now, by obliviousness, it is safe to reveal which memory addresses are accessed. We denote memory state of the program as mem with additive secret shares st_0, st_1 that Alice and Bob hold respectively.

Then Alice and Bob will emulate each step of the RAM program. For every step of the program, the known read address is addr. Alice reads $\mathsf{st}_0[\mathsf{addr}]$, and Bob reads $\mathsf{st}_1[\mathsf{addr}]$.

Then, they run a 2PC circuit protocol known as a *next-step instruction circuit*, where Alice's input is $\mathsf{st}_0[\mathsf{addr}]$, and Bob's input is $\mathsf{st}_1[\mathsf{addr}]$. The output of this protocol is $\mathsf{wdata}_0, \mathsf{wdata}_1, \mathsf{waddr}, \mathsf{raddr}$, where $\mathsf{wdata}_0, \mathsf{wdata}_1$ are secret shares of the write data for Alice and Bob, respectively, waddr is where the data is to be written, and raddr is the next read address.

Thus, this protocol will successfully implement garbled RAM, with the downside of increased round complexity. If the original program runs in time T, then using some practical ORAM, such as circuit ORAM, will have a $O(T \log^2 n)$ runtime.

3.2 Garbled RAM in Two Rounds

The main challenge of garbled RAM is that the garbler cannot predict which memory locations are accessed at each time step. To construct a garbled RAM protocol preserving Yao's round complexity, we will build tools to help us.

3.2.1 Garbled Memory

To work toward garbled RAM, we define an abstraction called *garbled memory*. Suppose that we want to support $\mathsf{read}(\mathsf{addr}), \mathsf{write}(\mathsf{addr}, \mathsf{data})$ and we allow ourselves to do this naively. That

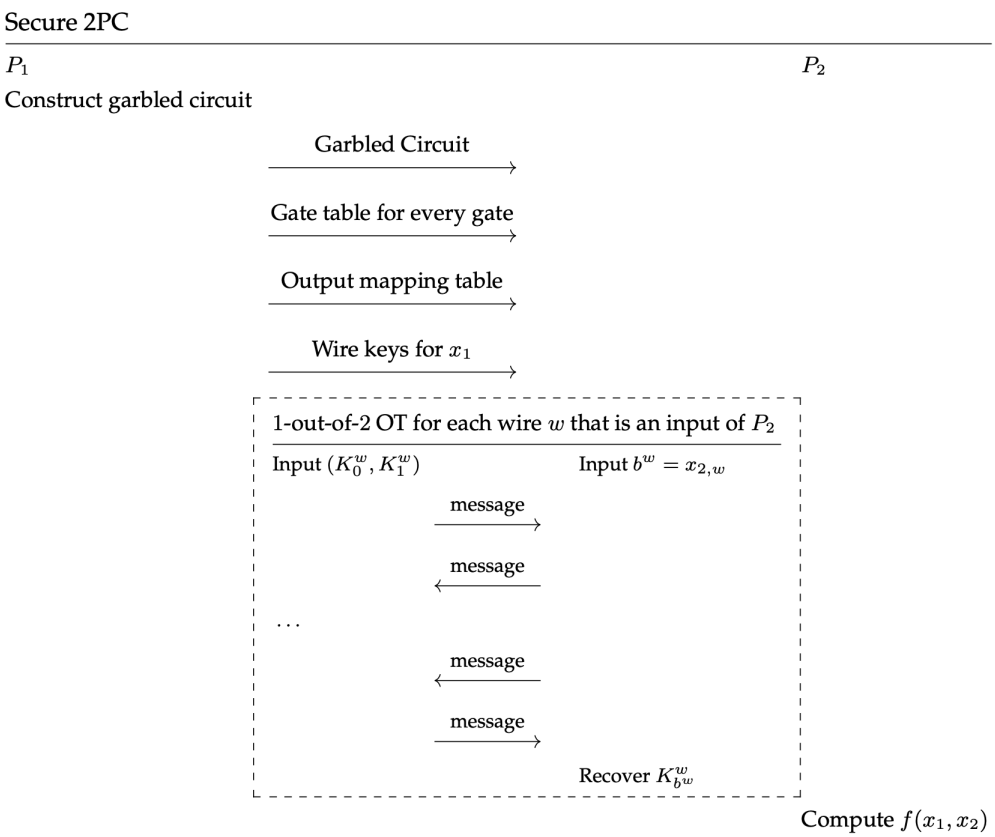

Figure 3: 2PC Protocol with Yao's Circuits

is, we are allowed to linear scan the memory to find and store the address to which we are reading/writing. In this case, the memory patterns are fixed since we just perform a linear scan every time. Also, this is easy to garble; say we have garbled memory under different labels $\mathsf{L}_{addr,t}$ in time step t (in garbled circuits, labels are the wire keys, but this language reframes the problem of propogating keys as a conceptually simpler "label translation"). The garbler can just generate all the labels and then the evaluator will know which label to use to read the garbled data. Although this incurs a linear cost to scan the garbled memory, this provides good intuition for the sublinear garbled RAM protocol.

3.2.2 Garbled Stack

Now, we work towards a more clever solution. Suppose that we want to garble a stack supporting two operations: $\mathsf{init}(\mathsf{D})$ and $\mathsf{pop}(\mathsf{b})$, where,

Stack Interface

- $\mathsf{init}(\mathsf{D})$: Initializes the stack with data D.
- $\mathsf{pop}(\mathsf{b})$: If $b = 0$, perform a fake pop (no state change) and output a dummy value. Otherwise, if $b = 1$, perform a real pop and return the top of the stack.

If the garbler knows ahead of time which pops are real, then garbling this stack is again easy and predictable. Unfortunately, this will not be the case in our use of the garbled stack.

To garble a stack when we cannot predict the pop patterns, we observe that our stack is just a special case of a turing machine, where the head of the stack simply moves down or stays put! Thus, we can use an oblivious turing machine (OTM) protocol as discussed in previous

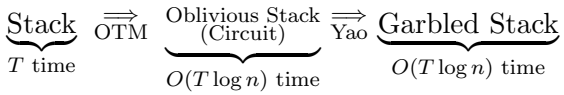

Figure 4: Two-Round Stack to Garbled Stack, with labeled run times

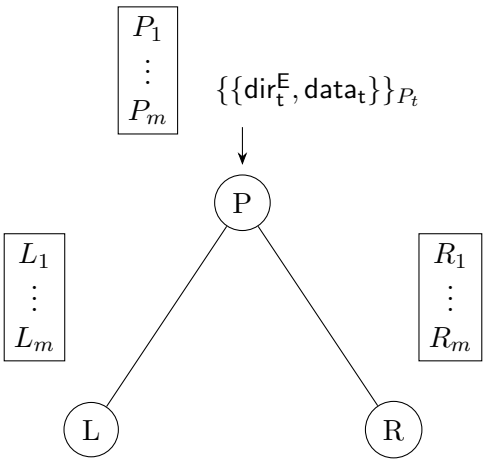

Figure 5: A Garbled Switch: Parent P receives $\{\{\mathsf{dir}_t^E, \mathsf{data}_t\}\}_{p_t}$ to route. Parent P has labels $P_1, \ldots, P_m$, left child L has labels $L_1, \ldots, L_m$, and right child R has labels $R_1, \ldots, R_m$.

lectures, which transforms the TM into a circuit with only $\log n$ factor overhead. That is, if our TM runs in time T with memory array size n, the circuit runs in time $O(T \log n)$.

3.3 Garbled Switch

The last gadget we will construct is the *garbled switch*. Our objective will be to garble a tree-based ORAM, like path-ORAM. However, we run into the issue that the access paths are unpredictable, similar to our previous garbling problems. The garbled switch will provide us with the mechanism of making the tree-based ORAM garbled by providing the label switching functionality.

Let $\{\{x\}\}_p$ denote x under a garbling with label p. The gadget will consist of a parent node and two child nodes.

At time step t, there is incoming data $\{\{\mathsf{dir}_t^E, \mathsf{data}_t\}\}_{p_t}$ where $\mathsf{dir}_t^E \in \{0, 1\}$ is the direction of routing (0 for left child, 1 for right child), data_t is the data to route. Here, we also use the $(.)^E$ superscript notation to denote that the variable does not need to be private. In the ORAM-tree, recall that it is ok for the direction to be known (and is necessary for efficiency).

The parent will hold labels $P_1, \ldots, P_m$, the left child will hold labels $L_1, \ldots, L_m$, and the right child will hold labels $R_1, \ldots, R_m$, where m is the maximum number of times the parent is invoked. The idea is that during the first time data is routed to the left node, it should be encoded under label L_1 ie $\{\{\mathsf{data}_t\}\}_{L_1}$, during the second time L_2 ie $\{\{\mathsf{data}_t\}\}_{L_2}$, and so on.... Thus, each node will keep track of a local time corresponding to each time it has been invoked so that we can apply a label that both parent and child agree with. Then, the child will be able to understand the garbled data and will be able to do its garbled computation on it.

We can construct a garbled switch protocol for the evaluator using two garbled stack to store corresponding labels,

Garbled Switch Protocol.

Suppose Evaluator has labels $L_1, \ldots, L_m, R_1, \ldots, R_m$ and garbling encoding function $\mathsf{garbenc}$:

1. Evaluator initializes two stacks:

$$\mathsf{stackL} \leftarrow \mathsf{GStack.init}(L_1, \ldots, L_m),$$
$$\mathsf{stackR} \leftarrow \mathsf{GStack.init}(R_1, \ldots, R_m)$$

2. In parent time step t, suppose the data is to be routed to the left child, whose local time is v. Evaluator runs:

$$\{\{\bar{L}\}\}_{P_t} \leftarrow \mathsf{StackL.pop}(\{\{1 - dir_t^E\}\}_{P_t})$$
$$\{\{\bar{R}\}\} \leftarrow \mathsf{StackR.pop}(\{\{dir_t^E\}\}_{P_t}),$$
$$\{\{data_t\}\}_{L_v} \leftarrow \mathsf{garbenc}(\{\{data_t\}\}_{P_t}, \{\{\bar{L}\}\}_{P_t})$$

Note that $\{\{\bar{R}\}\}$ is garbled under some garbage label.

The protocol uses a garbling encoding function $\mathsf{garbenc}$, which can be represented by an efficient garbled circuit, since its input and output are predictable (labels are only dependent on time). Likewise, the stacks are also predicatable, so there are no problems garbling.

3.3.1 Garbled Switches → ORAM Tree

With the garbled switch, we can combine many of them to make arbitrary trees. Our last step in garbling ORAM is to provide a procedure for reading and writing in this tree.

Suppose we have a path-ORAM tree, where every node implements a garbled switch and a constant-sized garbled bucket implemented with garbled memory (constant-size make linear scans cheap).

3.3.2 Garbled Reads

Recall in a nongarbled path-ORAM, if we already know the path denoted by leaf, along with read labels $\mathsf{RdL_0}, \ldots, \mathsf{RdL_D}$, where D is the depth, and the read address addr, we will search for addr along the path. If the element resides in the root bucket, then it will return the element garbled under $\mathsf{RdL_0}$. If the element is in the second node in the path, then it will return the element garbled under $\mathsf{RdL_1}$, and so on.... However, the next-instruction circuit will be garbled under the global time label (the root local time), so in order for the output to work with the next-instruction circuit, we also need to pass the read labels down the path, so every node will know what the output label should be.

Garbled path-ORAM Read Summary:

1. Garble the entire state of the read labels, path, and address along the path, so every node along the path can read the garbled data under the language it understands.
2. At each node along the path, check the bucket for the address and output either the answer or garbage under the correct output label.
3. Once the next-instruction circuit receives the $O(\log n)$ garbled results under the output label, it can decipher the real result and proceed to the next instruction.

3.3.3 Garbled Writes

To implement writing, we need to implement path-ORAM eviction. Recall that eviction takes place only along a path. We think of eviction in a different but equivalent way: suppose that every node along the path outputs some metadata. Then, for an eviction protocol, we have some circuit f taking in the metadata and outputting an eviction plan. For every node in the path, we use some circuit g that takes its bucket, eviction plan, and the data it has received and outputs its new bucket, eviction plan, and the data to pass on.

With this view, we can implement garbled eviction in the same way as garbled reads. We will pass down data along the path, every node will output some metadata garbled under some desired label (like the read labels), and the garbled metadata will be passed to a next-instruction circuit in a label it understands.

Garbled path-ORAM Write(Eviction) Summary:
Think of path-ORAM eviction as follows,

1. Suppose that every node along the path stores metadata, and that we have circuit f such that:
$$f(\mathsf{metadata}) \to \mathsf{evictionplan}$$
2. At each node along the path, we have circuit g outputting:
$$\begin{matrix}\mathsf{bucket}'\\ \mathsf{evictionplan}\\ \text{data to pass on}\end{matrix} \leftarrow g(\mathsf{bucket}, \mathsf{evictionplan}, \text{data recieved})$$
3. The next-instruction circuit takes the data and outputs the next instruction.

Then, the garbling is the same as reading.