2-Party Secure Computation: BetterYao

Ben Terner

1 Introduction

This survey will give an overview of the BetterYao implementation of secure computation against malicious adversaries, as described by [1] and [2] and implemented in [3]. This survey assumes familiarity with circuit garbling techniques, which are explained in another survey for reference.

1.1 Notation

We will discuss the protocol for securely evaluation a function $f(x, y) = (f_1(x, y), f_2(x, y))$ by two players P_1 and P_2, where P_1's input is x, P_1's output is f_1, P_2's input is y, and P_2's output is f_2. In our protocols, one of the players will take the role of *generator* and one will take the role of *evaluator*, or *Gen* and *Eval*, respectively. Without loss of generality, we can assume that P_1 will be *Gen* and P_2 will be *Eval*.

For each wire w_i in the circuit, *Gen* randomly picks two keys, $K_{i,0}$, $K_{i,1}$, and a permutation bit π_i. Each key is the length of the security parameter k. The *label* for each wire w_i consists of the pair $(K_{i,b}, b \oplus \pi_i)$ and is denoted $W_{i,b}$.

2 An Honest-But-Curious Protocol

We first describe a protocol for secure two-party computation by Yao that is secure in the honest-but-curious setting, and then explain potential attacks by malicious players and the mechanisms by which we enforce proper behavior or dampen the capabilities of attacks.

Gen will construct a circuit, gate by gate, according to Yao's protocol [4] and send each gate to *Eval*, as well as proper input keys for each of *Gen*'s inputs. *Eval* and *Gen* will use OTs in order for *Eval* to retrieve the proper keys for her own inputs. *Gen* also sends the random permutation bit π for each of the circuit output wires so that *Eval* can identify their semantic values.

So that *Eval* cannot trivially read the values of *Gen*'s output wires, *Gen* constructs the circuit to produce his output masked with one-time pad c. *Gen* keeps the value of c private and uses it to decrypt his output after receiving it from *Eval*, although keys for c must be also provided to *Eval*.

3 Attacks

3.1 Malicious Eval

The two-party protocol we describe is not *fair* in the sense that if *Eval* does not want to, she does not have to send *Gen* his outputs, but in this case *Gen* knows that *Eval* has cheated. *Eval* can behave maliciously against *Gen* in two other ways during this protocol: she can attempt to learn *Gen*'s outputs or she can report false outputs to *Gen*. We will refer to these as attacks against *Gen*'s *output privacy* and *Gen*'s *output authenticity.*

3.2 Malicious Gen

3.2.1 Generator's Input Consistency

Because we achieve security in the malicious protocol using Cut and Choose, *Gen* and *Eval* execute the Yao protocol on many circuits. *Gen* could attack *Eval* by providing inconsistent inputs to *Eval* in the evaluation circuits. Lindell and Pinkas [5] showed that for some functions, this could leak some information about *Eval*'s inputs.

3.2.2 Selective Failure

Gen could infer information about *Eval*'s inputs by providing a incorrect keys to *Eval* during OT that will force circuit decryption to fail. For example, a malicious *Gen* could assign keys (K_0, K_1) to one of *Eval*'s input wires when garbling a circuit but use (K_0, K_1^*) in the OT, where $K_1 \neq K_1^*$. If *Eval*'s input is 1, decryption of the first gate will fail and *Eval* will have to abort, indicating to *Gen* that her input was 1. If *Eval*'s input is 0, then *Gen* discovers her input by the knowledge that decryption of the circuit did not fail.

4 Security in the Malicious Setting

4.1 Cut and Choose

Before explaining defenses to specific concerns about *Gen* and *Eval* acting maliciously, we first discuss the *Cut and Choose* technique for circuit evaluation in the malicious setting. Intuitively, *Gen* will construct many circuits (the number determined by some security parameter) and send them to Eval, or at least commit to them. *Gen* and *Eval* will collaboratively choose some of the circuits at random to become "check circuits," with the rest being "evaluation circuits." *Gen* will reveal the private randomness used to construct the "check circuits," and *Eval* will verify their authenticity. *Eval* will evaluate the "evaluation circuits" as in the Yao protocol and select the output of the majority circuit as the protocol's output. However, for security purposes, *Gen* cannot know which circuits are check circuits and which are evaluation circuits, since if he did, he might change a circuit's output by switching the output keys for intermediate gates.

[1] showed that a 3:2 ratio of check circuits to evaluation circuits is near optimal (and better than 1:1).

Eval performs random seed checking. Explain the checks that Eval performs and what she has to do in order to perform the checks. Perhaps include a piece here about how some constructions compute the hash of a circuit.

4.2 Defenses Against Malicious Eval

4.2.1 Gen's Output Privacy

As explained in Section 2, *Gen*'s input privacy can be protected using a one-time pad circuit composed entirely of XOR gates. This does not change for the malicious setting.

4.2.2 Gen's Output Authenticity

At the end of circuit evaluation, *Eval* has outputs for both herself and *Gen*. Our protocol is not *fair* because *Eval* is not required to send *Gen* his outputs, but if *Eval* chooses to, she may attempt to deceive *Gen*. To convince *Gen* that *Eval* has been honest and to satisfy *Gen*'s concerns, they use a witness-indistinguishable proof.

Proof of Gen's Output Authenticity

Private Inputs

Gen has the output keys $\{(u_0^j, u_1^j)\}_{j\in[s]}$. *Eval* knows the index $m \in [s]$ of the majority circuit and the random key ν corresponding to *Gen*'s output wire of value a.

is $\nu = u_a^m$? This is the best I can tell.

Shared Inputs

Gen and *Eval* share the security parameter 1^k, statistical parameter 1^s, the commitments to *Gen*'s output keys $\{(com(u_0^j), com(u_1^{(j)}))\}_{j\in[s]}$, and *Gen*'s output a.

1. *Gen* chooses a random nonce $r \in \{0,1\}^k$ and he encrpyts it with each gate key $u_a^{(j)}$. He sends *Eval* the encryptions $enc_{u_a^{(1)}}(r)||enc_{u_a^{(2)}}(r)|| \cdots ||enc_{u_a^{(s)}}(r) = c^{(1)}||c^{(2)}|| \cdots ||c^{(s)}$

2. *Eval* commits to the decryption of $c^{(m)}$ and sends it to *Gen*. Formally, she sends $com(dec_v(c^m)) = com(r')$ to *Gen*.

3. *Gen* receives r' and then decommits to all of the keys $\{u_a^{(j)}\}_{j \in [s]}$

4. *Eval* checks *Gen*'s decommitments to determine if:

   (a) $com(u_a^j)$ correctly decommits to $u^{(j)}$ for $j \in [s]$

   (b) $dec_{u_a^{(i)}}(c^{(i)}) = dec_{u_a^{(j)}}(c^{(j)})$ for all $i, j \in [s], i \neq j$

   If any of the checks fails, *Eval* aborts. Otherwise, *Eval* decommits to r'

5. *Gen* checks that $com(r')$ decommits properly to r. If so, he accepts. Otherwise, he rejects.

4.3 Defense Against Malicious Gen

4.3.1 Gen's Input Consistency

The intuition to defend against this attack is to supplement our objective circuit with a 2-universal hash circuit that will compute some function over *Gen*'s inputs which *Eval* can verify for each circuit evaluation. It's critical that the hash function is both hiding and collision-free. Simply, to preserve the privacy property of the protocol, the output of the hash circuit should reveal no information to *Eval* about *Gen*'s inputs. Collision-freeness effectively binds *Gen* to his inputs; because two inputs are hard for *Gen* to find that will evaluate to the same hash, *Gen* must use the same semantic inputs for each garbled circuit that uses the same hash circuit (and because all garbled circuits use the same hash circuit, *Gen*'s input must be consistent). Two-Universal hash circuits satisfy the binding property by definition, since they fulfill the requirement that for fixed, distinct inputs x and y, the probability that a random hash function $h : A \to B$ satisfies $h(x) = h(y)$ is at most $1/|B|$.

It follows that if x_i is *Gen*'s input to evaluation circuit i, then the consistency of the hashes $h(x_1), h(x_2), ..., h(x_n)$ will imply the consistency of each x_i with probability at least $1 - 1/|B|$. Because the difficulty of finding collisions for 2-Universal hash functions is defined with *a posteriori* knowledge of the inputs x and y, the hash function must be chosen during the protocol *after Gen* commits to his inputs. Here, *Gen* commits to his input keys rather than his actual inputs in order to preserve privacy during the reveal phase.

It is not sufficient to simply add an auxiliary 2-Universal hash circuit to the protocol, since for circuits where *Gen* has few inputs, *Eval* can simply run all possible inputs by *Gen* through the hash circuit to find a matching hash and learn *Gen*'s inputs. In addition, the 2-Universal hash circuit must also be randomized (or, in a sense, salted). [2] use the Leftover Hash Lemma to show that *Gen* must pick $2k + lg(k)$ bits of fresh randomness at the beginning of the protocol as input to this hash function in order to achieve security according to parameter k, and that the output of the 2-Universal hash function will appear pseudorandom even if the hash function is made public.

The hash function is chosen from the family

$$\mathcal{M} = \{h_M | M \in \{0,1\}^{m \times n} \wedge h_M(x) = M \cdot x \text{ for some } m, n \in \mathbb{N}\}$$

which has the advantage that the hash circuit can be computed with only XOR-gates, making the computation overhead to the protocol minimal when using free-XOR.

4.3.2 Selective Failure

At a high level, the defense for this attack, first given by [5], is to provide a transformation that converts *Eval*'s true input y into the her protocol input $\overline{y}$, and have an auxiliary circuit convert $\overline{y}$ back into y during circuit evaluation. *Eval* does this by choosing some $M \in \{0,1\}^{n\times m}$ for some $m \in \mathbb{N}$ and computes $M \cdot \overline{y} = y$. This technique requires that the *Gen* be unable to infer any information about y from knowledge he may gain about $\overline{y}$. We require the following definition:

$M \in \{0,1\}^{n\times m}$ for some $n, m \in \mathbb{N}$ is called *k-probe-resistent* for some $k \in \mathbb{N}$ if for any $L \subset \{1,2,...,n\}$, the Hamming distance of $\oplus_{i\in L} M_i$ is at least k, where M_i denotes the *i-th* row of M.

If M is k-probe-resistent for some parameter k, then *Gen* will have negligible probability of inferring information about *Eval*'s input y from the protocol input $\overline{y}$, even if M is made public and computed exclusively with XOR gates.

Lindell and Pinkas [5] point out that as long as m is big enough, the M will not be k-probe-resistant with negligible probability. Although [5] choose m to be $max(4n, 8k)$, [2] give a probabilistic algorithm that produces k-probe-resistant matrix M such that $m \leq lg(n) + n + k + max(lg(4n), lg(4k))$. The algorithm follows:

Input: Eval's input size n and security parameter 1^k
Output: k-probe-resistent matrix $M \in \{0,1\}^{n\times m}$ for some $m \in \mathbb{N}$
$t \leftarrow \lceil max(lg(4n), lg(4k)) \rceil$ // find the minimum t such that $2^t \geq k + (lg(n) + n + k)/t$
while $2^{t-1} > k + (lg(n) + n + k)/(t-1)$ **do**
  $t \leftarrow t - 1$
end while
$K \leftarrow \lceil (lg(n) + n + k)/t \rceil$
$N \leftarrow K + k - 1$
for $i \leftarrow 1$ **to** n **do**
  Pick $P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{K-1} x^{K-1}$, where $a_i \xleftarrow{\$} \mathbb{F}_{2^t}$
  $M_i \leftarrow [P(1)_2 || P(2)_2 || ... || P(N)_2]$ // where $P(j)_2$ denotes a t-bit row vector
end for
return M // $M \in \{0,1\}^{n\times m}$, where $m = Nt$

The algorithm constructs a k-probe-resistant matrix M by randomly picking polynomials $P_1, P_2, ..., P_n \in \mathbb{F}_{2^t}[x]$ with degree at most K-1,where n is the evaluator's input size. The polynomials are evaluated at points $x_1, x_2, ...x_N \in \mathbb{F}_{2^t}$, and the outputs for each P_i over the points are concatenated as a $N \cdot t$-bit vector which becomes the i-th row of M.

This reduces the problem of probe resistance to Reed Solomon error correction codes.

discuss Reed Solomon

5 Performance Considerations

This section details a couple of engineering improvements to reduce the amount of computation and communication involved in the protocol.

5.1 Random Seed Checking

Communication cost of check circuits is independent of circuit size.

Recall that in the cut-and-choose paradigm, *Gen* and *Eval* must agree on some number of check circuits that *Eval* will verify.

5.2 Pipelining Evaluation

When circuits become millions or billions of gates large, it becomes infeasible to retain the entire circuit in memory, especially when multiple circuits must be evaluated to attain security in the malicious setting. HEKM [6] showed that holding entire circuits in memory is unnecessary, as *Gen* and *Eval* can execute the protocol while only retaining gates in memory that they need at the moment. To do this, *Gen* and *Eval* evaluate all of the circuits in lockstep and pipeline the garbling of gates with their evaluation. *Gen* garbles σ gates at once, all corresponding to the same gate in the evaluation circuit, and sends them to *Eval* together. While *Gen* is garbling, *Eval* evaluates the last batch that *Gen* sent.

explain issues with random seed checking and hash construction

5.3 Gate Communication

Recall that 60% of the circuits in our protocol are check circuits, but *Gen* does not know which circuits are check circuits and which are evaluation circuits. We'd like to find a way to prevent *Gen* from having to send all of the garbled gates over the communication channel when he only really needs to send 40% of them. Let the statistical security parameter used to decide the number of evaluation circuits be σ, and let μ be the number of check circuits. *Gen* should only have to communicate $\sigma - \mu$ pieces of information, one per evaluation circuit, in order for *Eval* to infer all of the garbled gates $\{g_0, g_1, \ldots, g_{\sigma-1}\}$.

To do this, *Gen* considers all of the garbled gate descriptions $\{g_0, g_1, \ldots, g_\sigma\}$ to be coefficients of a polynomial,

$$P(x) = g_0 + g_1 x + g_2 x^2 + \cdots + g_{\sigma-1} x^{\sigma-1}$$

and sends *Eval* $\sigma - \mu$ points of the polynomial $P(1), P(2), \ldots, P(\sigma - \mu)$. *Eval* can now use the coefficients she generates from the check circuits and polynomial interpolation to recover the gates she does now know. This technique sacrifices this small amount of computation overhead to save 60% of the communication cost.

6 Building Blocks and Assumptions

The protocol described in section 7 was presented by [2] and requires minimal assumptions, needing only a symmetric encryption scheme, oblivious transfer, and a commitment scheme. As a result, the strongest assumptions introduced in the protocol will result from our garbling techniques. Here, we present short descriptions of the building blocks we use in BetterYao and the assumptions they require.

6.1 Symmetric Encryption Scheme

The symmetric encryption scheme we use is relatively simple, but it also introduces the strongest assumption to our model. We consider AES to be a pseudorandom permutation and use the AES_NI instruction set to improve performance. As in [3], we use the encryption function $\mathrm{Enc}^k_{X,Y}(Z) = \text{AES-256}_{X||Y}(k) \oplus Z$, where k is the index of the garbled gate.

6.2 OT

We use OT by [?] , (also used by Lindell and Pinkas [7]?) and can conduct all of our OTs for inputs in parallel. The protocol is between a sender S and a receiver R, where S has inputs (k_0^i, k_1^i) for all i OT executions, and R has b^i to select k_0 or k_1 for every i. The parties share the description of a cyclic group $\mathbb{G}$ of order q, where q is the length of the security parameter 1^n, along with its generator g_0 (which, in fact, R selects and shares).

1. R randomly selects values $y, a \leftarrow \mathbb{Z}_q$ and $g_0 \leftarrow \mathbb{G}$, then sets $g_1 = (g_0)^y$. He computes $h_0 = (g_0)^a$ and $h_1 = (g_1)^{a+1}$.
   R sends the values (g_0, g_1, h_0, h_1) to S.

2. This step is a Zero Knowledge Proof that the information sent in step 1 is not a Diffie-Helman tuple (but $(g_0, g_1, h_0, \frac{h_1}{g_1})$ is), but our protocol has left it unimplemented.

3. R randomly selects $r^{(i)}$ for each OT instance i, and computes $gr^{(i)} = (g_{b_i})^{r^{(i)}}$ and $hr^{(i)} = (h_{b_i})^{r^{(i)}}$.
   R sends the values $gr^{(i)}, hr^{(i)}$ for all i to S.

4. S randomly chooses the values s_0, s_1, t_0, t_1 and computes $X_b = (g_b)^{s_b} \cdot h_b^{t_b}$ and $Y_b = K_b \cdot (gr_b)^{s_b} \cdot (hr_b)^{t_b}$ for $b \in \{0, 1\}$
   Then S sends R the values (X_0, X_1, Y_0, Y_1) for all i.

5. R selects X_b and Y_b from the choices of (X_0, X_1, Y_0, Y_1) and computes her outputs, $K_b = \frac{Y_b}{(X_b)^{r_i}}$.

In the code, this protocol randomly selects Y_0 and Y_1 before computing $Y_b = Y_b \cdot (gr_b)^{s_b} \cdot (hr_b)^{t_b}$ and does not include any of Sender's inputs. The comments indicate that Y_0 and Y_1 are chosen uniformly at random, as the keys are. In fact, our current code uses the OT results as seeds to pseudo-random number generators which are then used to mask Gen's communication with Eval about the things he should be sending. When Eval has the proper seed (corresponding to check circuit or evaluation circuit) she has the key for Gen's one-time pad encryption of the information he sends. When Eval does not have the seed, she cannot decrypt. Gen sends all of the check and evaluation information to Eval for each circuit, and Eval only learns some of it, as enforced by the OT.

6.3 Commitment Scheme

The commitment scheme we use is simply $hash(r' + \alpha)$, where r' is newly generated random bits and α is the value to which we commit. Decommitment requires sending $r' + \alpha$, and checking a commitment requires the receiver to compute the hash herself. Our hash function is SHA256.

6.4 Garbling Techniques

In our implementation, we use both Free-XOR [8] and GRR3 [9]. (We should also have some code for GRR2). Free-XOR and GRR3 are compatible when the hash function used to garble gates is circular 2-correlation robust [9, 10]. (Specifically, it is the Free-XOR technique that requires circular 2-correlation robustness.) We could also eliminate Free-XOR and use GRR2, which does not require the hash function to be correlation-robust. Implementing FleXOR [11] or Half-Gates [12] is future work.

6.5 Randomness, Random Seeds, and PRNGs

We briefly note that *Gen* and *Eval* both generate randomness in the protocol, and at some points *Gen* shares random seeds with *Eval*. We assume that *Gen* and *Eval* can find randomness from sufficiently random pools, and that when *Gen* shares random seeds, he and *Eval* have agreed on the PRNG which they seed in order to recover more bits. The protocol also relies on whatever assumptions the PRNGs make.

7 A Protocol Secure Against Malicious Adversaries

We now describe the full protocol for [fast] secure two-party computation in the malicious setting.

Private Inputs: *Gen*'s private inputs to the protocol are $x_i \in x$ and *Eval*'s private inputs to the protocol are $y_i \in y$.

Shared Inputs: *Gen* and *Eval* agree upon a function $f : (x, y) \rightarrow (f_1, f_2)$. *Gen* and *Eval* also agree on a security parameter 1^k and a statistical parameter 1^σ. They use a commitment scheme *com* and a symmetric encryption scheme (enc, dec).
Notation: *Gen*'s inputs $\overline{x}$ (described in step 1) have length m_1 and *Eval*'s inputs $\overline{y}$ (described in step 1) have length m_2. Consider any input key $K_{b,i}^{(j)} \in \{0,1\}^k$. K has length k and semantic value $b \in \{0,1\}$, is a key for the ith wire in its circuit, and belongs to circuit j. Let $W_{i,b}^{(j)}$ be the *label* corresponding to the ith wire w in circuit j such that $W_{i,b}^{(j)}$ has semantic value b. ($W_i^{(j)}$ has unknown semantic value).

1. **Input Modification**
   Gen generates randomness $r \in \{0,1\}^{2k+\lg(k)}$, which will be used as input to the 2-Universal circuit described in section 4.3.1. *Gen* also generates a one-time pad e which will be used to mask his outputs, as described in section 4.2.1. *Eval* computes her k-probe-resistant matrix M (as described in section 4.3.2) and input $\overline{y}$ such that $M \cdot \overline{y} = y$. *Gen*'s input is now $\overline{x} = x||e||r$ and *Eval*'s input is now $\overline{y}$.

2. **Gen Randomly Generates Input Keys**
   Gen generates randomness $\{\rho^{(j)}\}_{j\in\sigma}$, where $\rho^{(j)}$ corresponds to the randomness used for the jth circuit. He uses each $\rho^{(j)}$ to generate input keys and permutation bits $(K_{0,i}^{(j)}, K_{1,i}^{(j)}, \pi_i^{(j)}) \in \{0,1\}^{2k+1}$ for $i \in \overline{x}$ for each circuit $j \in \sigma$.

   At this point, *Gen* uses ρ only for his own input keys, so all of the bits used here can be drawn from true randomness. We can use a PRNG later to generate randomness later during circuit creation.

3. **Gen Commits to His Input**
   Gen generates new randomness $\gamma_i^{(j)}$ (independent of $\rho^{(j)}$) for $i \in \overline{x}$ and $j \in \sigma$ and commits to all of the keys that correspond to his circuit inputs. He sends $\Gamma = \{com(W_{i,b}^{(j)}; \gamma_i^{(j)})\}_{i\in\overline{x}}$ to *Eval*.

4. **Agreement on the Objective Circuit**
   Eval announces M to *Gen* and then *Gen* and *Eval* run an interactive coin-flipping protocol to generate the two-universal circuit $H \in \{0,1\}^{k\times m_1}$ used to compute a hash over *Gen*'s inputs. They now both know the full objective circuit C to compute $g : (\overline{x}, \overline{y}) \rightarrow (\perp, (h, c, g_2))$ where $h = H \cdot \overline{x}$, $c = g_1 \oplus e$, $g_1 = f_1(x, M \cdot \overline{y})$, and $g_2 = f_2(x, M \cdot \overline{y})$.

   The coin-flipping protocol is by [**?**]:

   (a) *Gen* and *Eval* generate random bits ρ_g and ρ_e
   (b) *Gen* sends *Eval* the commitment to his random bits, $com(\rho_g)$
   (c) *Eval* receives *Gen*'s commitment and replies with her own coins, ρ_e
   (d) *Gen* sends *Eval* the decommitment to his coins
   (e) *Eval* checks to ensure *Gen*'s decommitment is consistent with his random coins and aborts if not
   (f) *Gen* and *Eval* compute $\rho = \rho_g \oplus \rho_e$

   Here, we set the length of ρ to be $k \cdot m_1$, and ρ completely determines the two-universal circuit H that computes h by matrix multiplication.

5. **Gen Commits to Input and Output Labels**
   Gen uses $\rho^{(j)}$ to generate input keys for *Eval*'s inputs and output keys for his own outputs.

   Gen generates entire circuits here in order to derive his output keys, then throws them away (to save memory) and regenerates them again later.

   He then sends $(\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)})_{j\in\sigma}$ to *Eval*, where Θ represents *Gen*'s input labels, Ω represents *Eval*'s input labels, and Φ represents *Gen*'s output labels (specifically, the values of h and c):

(a) $\Theta^{(j)} = \{com(W^{(j)}_{i,0\oplus\pi^{(j)}_i};\theta^{(j)}_i), com(W^{(j)}_{i,1\oplus\pi^{(j)}_i};\theta^{(j)}_i)\}_{i\in m_1}$ where θ is randomness used in the commitment.

Do the θs in the above (which represent randomness for the commitment) need to be independent?

To protect *Gen*'s input privacy, the labels for each wire's 0 input and 1 input are permuted by (re)using the permutation bit π.

(b) $\Omega = \{com(W^{(j)}_{i,0};\omega^{(j)}_i), com(W^{(j)}_{i,1};\omega^{(j)}_i)\}_{i\in m_2}$ where ω is randomness used in the commitment. Unlike *Gen*'s inputs labels, *Gen* does not permute *Eval*'s input labels. She will need to know their ordering, and her inputs will be protected by OT in step 6.

(c) $\Phi^{(j)} = \{com(W^{(j)}_{i,0}), com(W^{(j)}_{i,1})\}_{i\in c}$
(Implicitly, this is $\Phi^{(j)} = \{com(W^{(j)}_{i,0}, \Phi^{(j)}_i), com(W^{(j)}_{i,1}, \Phi^{(j)}_i)\}_{i\in c}$, since all commitments require randomness.)

Find info about the commitment scheme we use (section 6.3).

These will be re-used in step 12, and *Eval* needs to know their semantics, so *Gen* does not permute them.

6. **Eval's Input OTs**
For every $i \in \overline{y}$, *Gen* and *Eval* perform $\binom{2}{1}$ OTs for *Eval*'s input, where *Gen*'s input is $(\{W^{(j)}_{i,0}, W^{(j)}_{i,1}\}_{j\in\sigma})$ and *Eval*'s input is $\overline{y}_i$. For each semantic value of $\overline{y}_i$, *Gen* sends the concatenation of the entire set of input keys over all j circuits.
We denote the set of decommitments that *Eval* receives for each circuit as $Y^{(j)} = \{(W^{(j)}_{i,\overline{y}_i}, \omega^{(j)})\}$.

7. **Cut and Choose**
Eval randomly chooses $S \subset [\sigma]$ such that $S = 2\sigma/5$. Use the string $s \in \{0,1\}^\sigma$ to describe the circuits that *Eval* has chosen for cut-and-choose by denoting $s_j = 1$ if $j \in S$ and $s_j = 0$ otherwise. *Gen* and *Eval* perform cut-and-choose by doing σ $\binom{2}{1}$ OTs, where *Eval*'s input is s_j and *Gen*'s input is $(\rho^{(j)}, X^{(j)})$ such that $X^{(j)} = X^{(j)}_1 \cup X^{(j)}_2$, where $X^{(j)}_1 = \{(W^{(j)}_{i,\overline{x}_i}, \gamma^j_i)\}_{i\in\overline{x}}$ and $X^{(j)}_2 = \{(W^{(j)}_{i,\overline{x}_i}, \theta^j_i)\}_{i\in\overline{x}}$. In other words, if *Eval* chooses a circuit as a check circuit, she learns the input and generates it. If *Eval* chooses a circuit as an evaluation circuit, she learns the de-commitments to *Gen*'s input keys (which will be checked) and can evaluate the circuit.

are Gen's inputs to the OT the same length? Do we need padding?

In our protocol implementation (BetterYao4 code), *Eval* and *Gen* choose the check circuits collaboratively. First, *Gen* and *Eval* interactively flip coins for a string of length k (the security parameter), and then they use the output to seed a pseudo-random number generator. They use this output to perform a Fisher-Yates shuffle of σ circuits and achieve a 3:2 ratio of check circuits to evaluation circuits by selecting the first $\frac{2\sigma}{5}$ of the circuits as evaluation circuits.

We also have the first circuit always being an evaluation circuit. Not sure why?

In the revision, we want cut-and-choose to be performed independently of *Gen*, meaning he does not get to know which circuits are check circuits, and we will use OTs as described above to enforce this privacy.

Note: the OTs in 7 can all be run in parallel, as can the OTs in step 6, and they can be run in parallel with each other.

8. **Circuit Garbling**
For every garbled gate $g : \{0,1\} \times \{0,1\} \to \{0,1\}$ with input wires w_a, w_b and output wire w_c, *Gen*

computes the garbled truth table:

$$G(g)^{(j)} = (< \pi_a^{(j)}, \pi_b^{(j)} >, < \pi_a^{(j)}, 1 \oplus \pi_b^{(j)} >, < 1 \oplus \pi_a^{(j)}, \pi_b^{(j)} >, < 1 \oplus \pi_a^{(j)}, 1 \oplus \pi_b^{(j)} >)$$

where $(< h_\alpha, h_\beta >) = enc_{K_{a,h_\alpha}^{(j)}}(enc_{K_{b,h_\beta}^{(j)}}(W_{c,g(h_\alpha,h_\beta)}^{(j)}))$.

come back to this equation.

Gen sends $\{G(C)^{(j)}\}_{j\in\sigma}$ to *Eval*, where $G(C)^{(j)} = \left(\{G(g)^{(j)}\}_{g\in C}, \{\pi_i^{(j)} : w_i \text{ is an output wire }\}\right)$

As explained in section 5.3, *Gen* does not need to send all of the gates he generates to *Eval*, but instead allows *Eval* to generate some of the gates on her own. *Gen* then encodes all of the gates that he generates (he still generates gates for every circuit) as coefficients of a polynomial and sends $\sigma - \mu$ points to *Eval*, allowing her to use the gates she knows from her μ check circuits to interpolate the polynomial and recover the remaining $\sigma - \mu$ gates.

9. **Checking Garbled Circuits**
   Eval must verify both check circuits and evaluation circuits.

   (a) **Check Circuits**
   For every $j \in [\sigma]\backslash S$, *Eval* uses $\rho^{(j)}$ to regenerate $\{\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)}\}$ received in step 5 and reconstruct $G(C)^{(j)}$.

   Here we note that because *Gen* sent *Eval* enough information during the garbling phase to recover all of the gates created at each step, *Eval*'s polynomial interpolation of each gate is an implicit check on $G(C)^{(j)}$. *Gen*'s only opportunity to lie about the check circuits is during the circuit OT in step 7, and if he lies about them then *Eval* will certainly discover it and abort, so in order for *Gen* to cheat on the evaluation circuit-gates (or the points on the polynomial sent at each step), he must be able to manipulate the gate generation function such that his input and output label commitments check out at the end of the protocol while the gates in between do not. formal proof?

   (b) **Evaluation Circuits**
   For every $j \in S$, *Eval* checks:

   i if the ith entry of $X_1^{(j)}$ received in step 7 successfully decommits the ith entry in $\Gamma^{(j)}$ received in step 3.

   ii if the ith entry of $X_2^{(j)}$ received in step 7 successfully decommits the $(2 \cdot i + \overline{x}_i \oplus \pi_i^{(j)})$-th entry of $\Theta^{(j)}$ received in step 5.

   iii if the decommitted labels from the above two checks are consistent with each other.

   iv if the set of *Eval* inputs $Y^{(j)}$ received in step 6 is consistent with half of the commitments in $\Omega^{(j)}$ received in step 5. Specifically, the ith entry of $Y^{(j)}$ should decommit the $(2 \cdot i + \overline{y}_i)$th entry in $\Omega^{(j)}$.

   If any failure occurs, *Eval* aborts.

10. **Evaluating Garbled Circuits**
    Eval evaluates the circuit according to the Yao protocol.

    (a) For every gate $g \in G(C)$ with input labels $W_a^{(j)} = (K_a^{(j)}, \delta_a^{(j)})$ and $W_b^{(j)} = (K_b^{(j)}, \delta_b^{(j)})$, *Eval* finds the $(2 \cdot \delta_a^{(j)} + \delta_b^{(j)})$ index E of $G(C)$ and computes

    $$W_c^{(j)} = (K_c^{(j)}, \delta_c^{(j)}) = dec_{K_b^{(j)}}(dec_{K_a^{(j)}}(E))$$

    (b) For every output wire w_i with label $W_i = (K_i, \delta_i)$, *Eval* computes the wire's value $b_i^{(j)} = \delta_i^{(j)} \oplus \pi_i^{(j)}$, where *Eval* learned $\pi_i^{(j)}$ at the end of step 8. She lets the set of outputs $\{b_i^{(j)}\}$ be the circuit outputs.

As described in section 5.2, *Eval* executes this stage and stage 9 in parallel with step 8. As *Gen* generates a new batch of gates, he sends them to *Eval* so that she can perform the proper interpolation, checks, and evaluation.

11. **Finding the Majority Output**
Eval finds the most commonly occurring element for each index in $\{b_i\}$ among the evaluation circuits and interprets the set of outputs $\{b_i\}$ as (h, c, g_2). She then checks:
    (a) if $h^{(j)} \neq h$ for any $j \in S$, or
    (b) if (h, c, g_2) is not the majority output of $\{(h^{(j)}, c^{(j)}, g_2^{(j)})\}$. More formally, *Eval* checks if
$$\left\{(h^{(j)}, c^{(j)}, g_2^{(j)}) : (h^{(j)}, c^{(j)}, g_2^{(j)}) = (h, c, g_2)\right\} \leq \frac{|S|}{2} = \frac{\sigma}{5}$$
If any of the above checks are true, *Eval* aborts. Otherwise, she accepts g_2 as her own output.

12. **Proving Gen's Output Authenticity**
Eval sends *Gen* his output c and must prove his output authenticity without revealing the index of the chosen majority circuit, as described in section 4.2.2.

References

[1] C.-h. Shen and A. Shelat, "Two-output secure computation with malicious adversaries," in *Advances in Cryptology–EUROCRYPT 2011.* Springer, 2011, pp. 386–405.

[2] C.-h. Shen and a. shelat, "Fast two-party secure computation with minimal assumptions," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM, 2013, pp. 523–534.

[3] B. Kreuter, a. shelat, and C.-H. Shen, "Billion-gate secure computation with malicious adversaries." in *USENIX Security Symposium*, 2012, pp. 285–300.

[4] A. YAO, "Protocols for secure computation," *23rd FOCS, 1982*, 1982.

[5] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *Advances in Cryptology-EUROCRYPT 2007.* Springer, 2007, pp. 52–78.

[6] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 35–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028102

[7] Y. Lindell and B. Pinkas, "Secure two-party computation via cut-and-choose oblivious transfer," Cryptology ePrint Archive, Report 2010/284, 2010, http://eprint.iacr.org/.

[8] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *Automata, Languages and Programming.* Springer, 2008, pp. 486–498.

[9] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," in *Advances in Cryptology–ASIACRYPT 2009.* Springer, 2009, pp. 250–267.

[10] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou, "On the security of the "free-xor" technique," in *Theory of Cryptography.* Springer, 2012, pp. 39–53.

[11] V. Kolesnikov, P. Mohassel, and M. Rosulek, "Flexor: Flexible garbling for xor gates that beats free-xor," in *Advances in Cryptology–CRYPTO 2014.* Springer, 2014, pp. 440–457.

[12] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," Cryptology ePrint Archive, Report 2014/756, 2014, http://eprint.iacr.org/.