2-Party Secure Computation: A Survey of 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. There's a piece missing somewhere about how to 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.

3

4.3.2 Selective Failure

At a high level, he 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-resistent with negligible probability. Although [5] choose m to be $max(4n, 8k)$, [2] give a probabilistic algorithm that produces k-probe-resistent 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 $m = Nt$

> I still don't understand this. The coefficients are drawn from $\mathbb{F}_{2^t}$ and used to find points in $\mathbb{F}_{2^t}$, and then points in the polynomial from 0 to N are concatenated to get a point in $\mathbb{F}_{2^k}$? TODO: error correction codes.

The algorithm constructs a k-probe-resistent 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.

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 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

5.2 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 communication overhead to save 60% of the communication cost.

5.3 Random Seed Checking

> Communication cost of check circuits is independent of circuit size.

6 Building Blocks

6.1 Symmetric Encryption Scheme

6.2 OT Extension

6.3 Commitment Scheme

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 $f : (x, y) \to (f_1, f_2)$. *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 $j \in \sigma$.

   > How much randomness does *Gen* need, or equivalently, how long should the seed ρ be?

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 $g : (\overline{x}, \overline{y}) \to (\bot, (h, c, g_2))$ where $h = H \cdot \overline{x}$, $c = g_1 \oplus e$, $g_1 = f_1(x, M \cdot \overline{y})$, and $g_2 = f_2(x, M \cdot \overline{y})$.

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.

   > Does *Gen* generate entire circuits here, or are these keys enough? It should be implied that *Gen* must actually generate the whole circuit in order to get the output keys.

   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 c):

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

   > Do the θs in the above (which represent randomness for the commitment) need to be separate, or even 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) $\Omega = \{com(W_{i,0}^{(j)}; \omega_i^{(j)}), com(W_{i,1}^{(j)}; \omega_i^{(j)})\}_{i \in m_2}$ 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) $\Phi = \{com(W_{i,0}), com(W_{i,1})\}_{i \in c}$

   > Does Φ not require randomness in its commitments? Why?

   Eval will discover the semantics of these wires anyway (in step ??), so there is no need to 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 $(\{W_{i,0}^{(j)}, W_{i,1}^{(j)}\}_{j \in \sigma})$ 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 $Y^{(j)} = \{(W_{i,\overline{y}_i}^{(j)}, \omega^{(j)})\}$.

6

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 $X^{(j)} = X_1^{(j)} \cup X_2^{(j)}$, where $X_1^{(j)} = \{(W_{i,\overline{x}_i}^{(j)}, \gamma_i^j)\}_{i \in \overline{x}}$ and $X_2^{(j)} = \{(W_{i,\overline{x}_i}^{(j)}, \theta_i^j)\}_{i \in \overline{x}}$. 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?

   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 $G(C)^{(j)} = (\{G(g)^{(j)}\}_{g \in C}, \{\pi_i^{(j)} : w_i$ is an output wire $\})$

   > talk about using optimizations described above in communication (and talk about them!)

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 $G(C)^{(j)}$.

   (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.

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.

> There will be a version where Gen has to generate all the circuits twice (so as not to keep them in memory unnecessarily). Make this inclusion in the above list. Also, include the part about cutting down communication.

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