Efficient Three-Party Computation from Cut-and-Choose

Seung Geol Choi*† Jonathan Katz‡ Alex J. Malozemoff‡ Vassilis Zikas§†

Abstract

With relatively few exceptions, the literature on efficient (practical) secure computation has focused on secure two-party computation (2PC). It is, in general, unclear whether the techniques used to construct practical 2PC protocols—in particular, the *cut-and-choose* approach—can be adapted to the multi-party setting.

In this work we explore the possibility of using cut-and-choose for practical secure *three-party* computation. The three-party case has been studied in prior work in the semi-honest setting, and is motivated by the observation that real-world deployments of multi-party computation are likely to involve few parties. We propose a constant-round protocol for three-party computation tolerating any number of malicious parties, whose computational cost is essentially only *twice* that of state-of-the-art two-party protocols.

1 Introduction

The past few years have seen a tremendous amount of attention devoted to making secure computation truly practical (e.g., [HEKM11, KsS12, KSS13]). With only a few exceptions [DPSZ12, DKL$^+$13, KSS13], however, this work has tended to focus on secure *two-party* computation (2PC). In the semi-honest setting, a series of papers [BDNP08, HKS$^+$10, HEKM11, HEK12] showed that Yao's garbled circuit technique [Yao86] can yield very efficient protocols for the computation of boolean circuits. In the malicious setting, Lindell and Pinkas [LP07] initiated use of the *cut-and-choose* technique, also based on Yao's garbled circuits, for constructing efficient, constant-round protocols. This technique was developed further in several subsequent works [Woo07, LPS08, NO09, PSSW09, LP11, sS11, KsS12, HKE13, Lin13, MR13, sS13], and yields the fastest known protocols for (malicious) secure two-party computation (2PC) of boolean circuits.

2PC protocols with malicious security can also be based on the GMW protocol [GMW87] (e.g., [NNOB12]). Although this approach yields protocols with round complexity linear in the (multiplicative) depth of the circuit, it offers the advantage that much of the computation can be pushed to an *offline*, pre-processing phase that is executed before the parties receive their inputs. The subsequent, *online* computation is very fast and uses mainly information-theoretic techniques.

In the setting of *multi*-party computation (MPC) with security against an arbitrary number of corruptions, the situation is somewhat different. While there has been much recent work on

*United States Naval Academy. **Email:** choi@usna.edu

†Portions of this work were done while at the University of Maryland.

‡Dept. of Computer Science, University of Maryland. **Email:** {jkatz,amaloz}@cs.umd.edu

§University of California, Los Angeles. **Email:** vzikas@cs.ucla.edu

optimizing MPC for *semi-honest* adversaries [BDNP08, BLW08, BCD+09, DGKN09, BSMD10, BTW12, CHK+12], less work has focused on security against malicious corruptions. The work of Ishai, Prabhakaran, and Sahai [IPS08] gives protocols with good *asymptotic* efficiency; however, despite some promising optimizations [LOP11], it has not yet produced practical instantiations. The SPDZ protocol [BDOZ11, DKL+12, DPSZ12, DKL+13, KSS13], which handles arithmetic circuits, has extremely fast online running time at the cost of a very slow offline phase.

However, unlike protocols based on garbled circuits, SPDZ runs for a linear (instead of constant) number of (online) rounds, and in each such round every party needs to utilize a broadcast channel. To our knowledge, SPDZ's implementation experiments [DKL+12, DPSZ12, DKL+13] were run on a local-area network where physical broadcast is available, and thus the delay due to accounting for round-timeouts and/or running a multi-party broadcasting protocol when operating in a wide-area network environment has not been taken into account. This delay may be non-trivial depending on circumstances: Schneider and Zohner [SZ13] have shown that as the latency between machines increases, the cost of each round becomes more and more significant.

MPC for a small number of parties. Research on secure computation has traditionally been divided into two classes: work focusing on two-party computation, and work focusing on multi-party computation for an arbitrary number of parties.[1] Yet, in practice, it seems that the most likely scenarios for secure MPC would involve a small number of parties. Indeed, the only implementations of MPC which have been deployed in practice are for *three* parties [BLW08, BCD+09]. (See also `https://sharemind.cyber.ee/clouddemo/` for an online demo of MPC using three parties.) In general, as the number of parties increases, the cost of communication amongst the parties increases as well. In a wide-area network setting, this may have a huge impact on the running time of the protocol. Motivated by these observations, in this work we initiate the study of efficient three-party computation in the malicious model, tolerating an arbitrary number of corruptions.

Our contributions. We construct the first practical, constant-round protocol for secure three-party computation of boolean circuits. Our protocol uses player-simulation techniques in order to compile (cut-and-choose-based) 2PC protocols (e.g., [LP07, LP11, Lin13]) into three-party protocols. We instantiate our compiler with state-of-the-art 2PC constructions and show that the addition of a third party comes at the cost of roughly a factor two overhead over the underlying 2PC protocol in terms of computation, and a factor three overhead in terms of communication. This running time appears to be superior to the state-of-the-art MPC protocols in terms of *start-to-finish* running time.[2] Of course, computing the exact overhead requires implementations of both our protocol and the underlying 2PC protocol and is a subject of future research. As a further optimization point, our protocol makes *only three calls overall* to a broadcast channel (one with each party as sender), as opposed to existing practical MPC solutions (for more than two parties) which use broadcast for communicating all protocol messages. This may be important in certain wide-area network settings where communication (and broadcast specifically) is very expensive.

Overview of our 3PC protocol. Denote the three parties by P_1, P_2, and P_3. The high-level idea of our construction is to execute a two-party protocol $\widehat{\pi}$, where one of the two parties (say $\widehat{P}_1$) is emulated by P_1 and P_2 via a two-party protocol π, and the other party is played by P_3.

[1] Here we are interested in protocols tolerating an arbitrary number of corruptions. One could further distinguish work on MPC that assumes an honest majority.

[2] We note, however, that much of our protocol can be pre-processed as well.

Clearly, naïvely applying the above idea yields an inefficient construction even when state-of-the-art 2PC protocols are used for π and $\widehat{\pi}$. Assume, for example, that the most efficient 2PC protocol is used for both π and $\widehat{\pi}$, where π simply computes the circuit of $\widehat{P}_1$ among P_1 and P_2. The security of the resulting construction follows trivially from the composition theorem. However, unless the size of the circuit is very small, this approach results in a huge blowup on the overall runtime; in particular, if t is the time π needs to compute the circuit of $\widehat{P}_1$ and $\widehat{t}$ is the time that $\widehat{\pi}$ needs to compute the three-party circuit, then the runtime of the above naïve construction is $t \cdot \hat{t}$. Taking into account that the computation of the parties in all known 2PC protocols (i.e., candidates for π) usually includes several encryption or decryption operations, t will typically be bigger that $\widehat{t}$, yielding at least a quadratic blowup.

Emulating the sender versus emulating the receiver. In most cut-and-choose-based 2PC protocols, the parties have distinct roles: one is the *sender*, or circuit generator, and the other is the *receiver*, or circuit verifier. One might be tempted to think that, because the role of the verifier in the protocol is more "passive" (in the sense that the computation is less complicated), the most natural approach would be to emulate the verifier among P_1 and P_2 (and have P_3 locally do the heavier work doing circuit generation and opening over broadcast). This seemingly direct approach fails as one needs a mechanism for P_1 and P_2 to include their inputs into the garbled circuits. Clearly, doing so by having P_1 first receive his input-keys via OT (as in the original Yao-based constructions) and then handing them to P_2 yields an insecure protocol; indeed, an adversary corrupting P_2 and P_3 can then trivially learn P_1's inputs.

Instead, in this work we have P_1 and P_2 emulate the sender, and we have P_3 play the role of the receiver. More precisely, we adapt the distributed circuit-garbling technique [BMR90, DI05] to the two-party setting, allowing P_1 and P_2 to compute a sharing of a garbled circuit which they then reconstruct for P_3. By appropriate optimizations, we ensure that distributed garbling requires P_1 and P_2 to compute and communicate roughly as much as the sender in an execution of the Yao protocol (plus some OT calls per gate); P_3 needs to do nothing during the circuit garbling. Most interestingly, our construction features a mechanism which allows P_3 to receive the keys corresponding to his input bits for evaluating the garbled circuit by only one invocation of OT per input-bit with each of P_1 and P_2. Thus, the total computational cost of the distributed garbling is roughly twice the cost of locally garbling the circuit, plus the cost of a small number of OTs per gate which can be efficiently amortized through the use of OT extension [IKNP03, NNOB12]. The communication cost of this distributed garbling is roughly equivalent to that of sending a garbled circuit.

Our distributed garbling scheme is secure against malicious adversaries, which ensures that an adversary corrupting only one of the parties P_1 and P_2 cannot produce a maliciously constructed garbled circuit. In order to protect against an adversary who corrupts both P_1 *and* P_2, we rely on the cut-and-choose technique. We give concrete instantiations (in the Random Oracle Model) of our protocol using a combination of two 2PC protocols by Lindell and Pinkas [LP07, LP11] and the more recent protocol by Lindell [Lin13] which drastically reduces the number of circuit garblings required for cut-and-choose.

Interestingly, the cut-and-choose technique does not only protect against corrupting both P_1 and P_2, but allows a considerable efficiency improvement. More precisely, it allows us to avoid using costly authenticated shares (towards P_3) for the computed (shared) garbled circuit. Instead, our distributed garbling scheme outputs, even in the malicious setting, a plain two-out-of-two sum sharing of the garbled circuit.

The security model. We assume the reader is familiar with the simulation paradigm [GMW87, Can00, Can01, Gol09]. For simplicity we restrict ourselves to computation of non-reactive functions, also known as Secure Function Evaluation. We consider a *static* adversary that gets to choose the corrupted parties at the beginning of the protocol. Without loss of generality, we assume that the circuit to be computed is deterministic and has a single output received by one of the parties. We point out that although we use the language of [Can00], all our proofs use straight-line black-box simulators, which makes our protocols secure in the Universal Composition framework of Canetti [Can01].

Outline. In Section 2 we cover preliminary topics. In Section 3 we describe our two-party distributed garbling scheme, and in Section 4 we discuss our three party protocol. In Section 5 we show how to instantiate the various two-party functionalities we utilize in our protocols.

2 Preliminaries

We let k denote the computational security parameter and let s denote the statistical security parameter. We use $x \xleftarrow{\$} S$ to denote choosing a value x uniformly at random from the set S, and use $\|$ to denote concatenation.

Circuit notation. We follow the circuit notation of [BHR12]. Let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$ be a circuit, where n is the number of input wires, m is the number of output wires, and q is the number of gates, where each gate is indexed by its output wire. Thus, the total number of wires in the circuit is $\mathsf{n} + \mathsf{q}$. The numbering of wires starts with the inputs and ends with the outputs; i.e., we have inputs $\{1, \ldots, \mathsf{n}\}$ and outputs $\{\mathsf{n} + \mathsf{q} - \mathsf{m} + 1, \ldots, \mathsf{n} + \mathsf{q}\}$. The function L (resp., R) takes as input a gate index and returns the left (resp., right) input wire to the gate. We require $L(\gamma) < R(\gamma) < \gamma$ for any gate index γ. The function G encodes the functionality of a given gate, e.g., $G_\gamma(0, 1) = 0$ if the gate with index γ is an AND gate. Because we consider circuits with inputs from multiple parties, let $\{\mathsf{n}_{i-1} + 1, \ldots, \mathsf{n}_i\}$ denote the input wires "controlled" by party P_i, with $\mathsf{n}_0 = 0$.

We denote *input gates* as those gates with one or more input wires, *inner gates* as those gates with no input or output wires, and *output gates* as those gates with an output wire.

Secret sharing. Our constructions use two-out-of-two secret sharing. In the semi-honest setting, we use a standard (linear) sharing of strings: the secret $x \in \{0,1\}^*$ is split into two random *summands* x_1 and x_2 such that $x_1 \oplus x_2 = x$, with P_i holding the summand x_i. We denote the *sharing* of x by $[x] = ([x]^{(1)}, [x]^{(2)})$, where we refer to each $[x]^{(i)} = x_i$ as P_i's *share* of x. This sharing is linear: If $[x]$ and $[y]$ are sharings of x and y respectively, then $[x] \oplus [y]$ is a sharing of $x \oplus y$; that is, $[x \oplus y] = [x] \oplus [y]$ and thus P_i can locally compute his share as $[x \oplus y]^{(i)} = [x]^{(i)} \oplus [y]^{(i)}$. It is straight-forward to verify that the above secret-sharing is *private* provided that the summands x_1 and x_2 are uniformly chosen (restricted only on $x_1 \oplus x_2 = x$); i.e., any single share $[x]^{(i)}$ contains no information about the secret x. Reconstructing a sharing $[x]$ is easily done by having each party announce his share $[x]^{(i)}$ and taking x to be the exclusive-or of the announced shares.

Our protocols use shares of two types of secrets: k-bit strings $x \in \{0,1\}^k$ and bits $b \in \{0,1\}$. For clarity in the presentation, we use the bracket notation introduced above for sharings of $x \in \{0,1\}^k$, and use the notation $\langle \cdot \rangle$ for sharings of bits; i.e., if $b \in \{0,1\}$ then a sharing of b is denoted as $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$.

In the malicious setting we need the sharings of bits to be *authenticated*; i.e., in addition to his summand b_i, each party P_i holds an authentication tag t_i for a Message Authentication Code (MAC), with another party P_j holding the corresponding verification key k_j. More precisely, in a sharing $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$ of b, each party's share is now a tuple $\langle b \rangle^{(i)} := (b_i, t_i, k_j)$, where $b_1 \oplus b_2 = b$, and t_i is a valid MAC on b_i with key k_j. This ensures that the adversary cannot make the reconstruction output any value other than the secret b. In particular, to reconstruct some sharing $\langle b \rangle = (\langle b \rangle^{(1)}, \langle b \rangle^{(2)})$, each party P_i first announces his summand b_i and the corresponding authentication tag t_i; subsequently, each party P_i checks that the other party P_j announced a validly authenticated summand matching his own verification key and if this is not the case he rejects. The inability of an adversarial P_i to announce a summand other than b_i follows from the unforgeability of the MAC, as P_i does not know the key k_j matching his authentication tag.

We also assume this authentication is linear in the following sense: Given $\langle b \rangle$ and $\langle b' \rangle$, the parties can compute $\langle b \rangle \oplus \langle b' \rangle$ *locally*. Namely, $\langle b \rangle \oplus \langle b' \rangle = (\langle b \oplus b' \rangle^{(1)}, \langle b \oplus b' \rangle^{(2)})$, where $\langle b \oplus b' \rangle^{(i)} = (b_i \oplus b'_i, t_i \oplus t'_i, k_j \oplus k'_j)$ is a valid authentication. We show how to construct such an authenticated sharing in Section 5.

3 Two-Party Distributed Garbling Scheme

In this section we describe our construction of a two-party distributed garbling scheme. Our protocol combines the standard Yao garbling circuit technique with the distributed garbling ideas from Damgård and Ishai [DI05]. The main idea is the following: The players jointly compute a garbled circuit, where the gates are garbled by use of a distributed encryption scheme which takes, for each encryption, one key from each party.

In more detail, our construction is described in several steps. As the first step, in Section 3.1 we give a description of our garbling scheme; i.e., the code of the sender in our version of Yao's protocol. This section gives the reader familiarity with our notation and is used as a reference in the distributed protocol. Next, in Section 3.2 we describe an efficient (semi-honest) protocol that allows parties P_1 and P_2 to securely emulate the circuit-garbling procedure from Section 3.1. Finally, in Section 3.3, we show how to make the garbling procedure maliciously secure.

3.1 Single-Party Garbling Scheme

Our garbling scheme is a slight variant of the protocol from [DI05] adapted to two parties. This should be regarded as an initial step towards our ultimate goal of a *distributed* garbling scheme. Here, we describe the high-level construction; see Figure 1 for the detailed protocol.

We associate two random keys $K_{w,0}, K_{w,1}$ with each wire w in the circuit; key $K_{w,0}$ corresponds to the value '0' and $K_{w,1}$ corresponds to the value '1'. Each key $K_{w,b}$ consists of two sub-keys $s^1_{w,b}$ and $s^2_{w,b}$; that is, $K_{w,b} = (s^1_{w,b}, s^2_{w,b})$. In addition, for each wire w we choose a random mask bit λ_w. Each key has an associated tag, derived from the mask bit, which acts as a blinding of the true value the key represents.

Now, consider gate G_γ in the circuit with input wires α and β. The garbled gate of G_γ consists of an array of four encryptions: for each $(b_\alpha, b_\beta) \in \{0,1\} \times \{0,1\}$, the row (b_α, b_β) consists of an encryption of $K_{\gamma, G_\gamma(b_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta) \oplus \lambda_\gamma}$ and its corresponding tag $G_\gamma(b_\alpha \oplus \lambda_\alpha, b_\beta \oplus \lambda_\beta) \oplus \lambda_\gamma$ under keys K_{α,b_α} and K_{β,b_β}. Let P denote a table that stores all the garbled gates; in particular, the entry $P[\gamma, b_\alpha, b_\beta]$ contains an encryption corresponding to row (b_α, b_β) of the garbled gate for G_γ.

Auxiliary Inputs: Security parameter k, circuit $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$.

1. **Generate masks:**
   - *Generate input and inner mask bits*: For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}-\mathsf{m}\}$: generate $\lambda_w \xleftarrow{\$} \{0,1\}$.
   - *Generate output mask bits*: For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: generate $\lambda_w \leftarrow 0$.
2. **Generate sub-keys:**
   - For wires $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}\}$ and $b \in \{0,1\}$: generate sub-keys $s^1_{w,b}, s^2_{w,b} \xleftarrow{\$} \{0,1\}^k$.
3. **Construct garbled circuit:**
   - *Construct garbled gates*: For gates $\gamma \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}\}$, do the following:
     Let $\alpha \leftarrow L(\gamma)$ and $\beta \leftarrow R(\gamma)$ be the index of the left and right input wires, respectively, of the gate indexed by γ. Letting $K_{w,b} = (s^1_{w,b}, s^2_{w,b})$, compute the following:

$$P[\gamma,0,0] \leftarrow \mathsf{Enc}_{K_{\alpha,0},K_{\beta,0}}\left(K_{\gamma,G_\gamma(\lambda_\alpha,\lambda_\beta)\oplus\lambda_\gamma} \| G_\gamma(\lambda_\alpha,\lambda_\beta)\oplus\lambda_\gamma\right)$$
$$P[\gamma,0,1] \leftarrow \mathsf{Enc}_{K_{\alpha,0},K_{\beta,1}}\left(K_{\gamma,G_\gamma(\lambda_\alpha,\lambda_\beta\oplus 1)\oplus\lambda_\gamma} \| G_\gamma(\lambda_\alpha,\lambda_\beta\oplus 1)\oplus\lambda_\gamma\right)$$
$$P[\gamma,1,0] \leftarrow \mathsf{Enc}_{K_{\alpha,1},K_{\beta,0}}\left(K_{\gamma,G_\gamma(\lambda_\alpha\oplus 1,\lambda_\beta)\oplus\lambda_\gamma} \| G_\gamma(\lambda_\alpha\oplus 1,\lambda_\beta)\oplus\lambda_\gamma\right)$$
$$P[\gamma,1,1] \leftarrow \mathsf{Enc}_{K_{\alpha,1},K_{\beta,1}}\left(K_{\gamma,G_\gamma(\lambda_\alpha\oplus 1,\lambda_\beta\oplus 1)\oplus\lambda_\gamma} \| G_\gamma(\lambda_\alpha\oplus 1,\lambda_\beta\oplus 1)\oplus\lambda_\gamma\right)$$

4. **Output circuit:**
   - Set $GC \leftarrow (\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, P)$, and output:

$$\left(GC, \left\{(s^1_{w,b\oplus\lambda_w}, s^2_{w,b\oplus\lambda_w}, b\oplus\lambda_w) : w \in \{1,\ldots,\mathsf{n}\}, b\in\{0,1\}\right\}\right).$$

Figure 1: Circuit garbling scheme.

Evaluation proceeds as follows. Let α and β be input wires connected to gate G with index γ. The evaluator is given $(K_{\alpha,b_\alpha\oplus\lambda_\alpha}, b_\alpha\oplus\lambda_\alpha)$ and $(K_{\beta,b_\beta\oplus\lambda_\beta}, b_\beta\oplus\lambda_\beta)$, along with P. He takes the row $P[\gamma, b_\alpha\oplus\lambda_\alpha, b_\beta\oplus\lambda_\beta]$ and decrypts it using the keys $K_{\alpha,b_\alpha\oplus\lambda_\alpha}$ and $K_{\beta,b_\beta\oplus\lambda_\beta}$, resulting in $(K_{\gamma,G(b_\alpha,b_\beta)\oplus\lambda_\gamma}, G(b_\alpha,b_\beta)\oplus\lambda_\gamma)$. It is straightforward to verify that by continuing this evaluation, the output of each gate will be revealed masked by its corresponding mask. By picking masks of the output wires to be '0' we ensure that the evaluator receives the (unmasked) output of the circuit.

3.2 Distributing the Garbling Scheme Between Two Parties

We now show how to emulate the above garbling scheme between two parties in the *semi-honest* setting. We assume the parties have access to the following two-party ideal functionalities:

- *Gate computation* $\mathcal{F}^G_{\mathbf{gate}}(\langle a\rangle, \langle b\rangle)$: The functionality takes as input sharings $\langle a\rangle$ and $\langle b\rangle$ of bits a and b, respectively, and is parameterized by a binary gate G; it outputs a sharing $\langle G(a,b)\rangle$ of the output of G on input (a,b).
- *One-out-of-two oblivious secret sharing* $\mathcal{F}^i_{\mathbf{oshare}}(\langle b\rangle, m_0, m_1)$: The functionality takes as input a sharing $\langle b\rangle$ of a bit b (i.e., each party inputs his share), along with two messages m_0, m_1 from P_i, and outputs a random two-out-of-two sharing $[m_b]$ of m_b.
- *Constant bit sharing* $\mathcal{F}^b_{\mathbf{const}}()$: The functionality is parameterized by a bit $b \in \{0,1\}$, and outputs a random sharing $\langle b\rangle$ of b.
- *Random bit sharing* $\mathcal{F}_{\mathbf{rand}}()$: The functionality chooses a random bit $r \xleftarrow{\$} \{0,1\}$ and computes and outputs a random sharing $\langle r\rangle$ of r.

– *Bit secret sharing* $\mathcal{F}_{\mathbf{ss}}^{i}(b)$: The functionality takes input bit $b \in \{0,1\}$ from P_i and outputs a random two-out-of-two sharing $\langle b \rangle$ of b.

Each of these can be instantiated efficiently in the semi-honest setting; see Appendix A and Appendix B for details.

Distributed encryption scheme. We utilize the distributed encryption scheme from [DI05]. Suppose the message and the key for the encryption scheme are distributed as follows:

– The message m is secret-shared; i.e., P_1 and P_2 hold $[m]^{(1)}$ and $[m]^{(2)}$, respectively.
– The encryption key $K = (s^1, s^2)$ is distributed such that P_1 and P_2 hold s^1 and s^2, respectively.

The encryption of the secret-shared message m with tweak T under key $K = (s^1, s^2)$ is:

$$\mathsf{Enc}_K^T(m) = (\mathsf{Enc}_{s^1,T}^1(m), \mathsf{Enc}_{s^2,T}^2(m)) = \left([m]^{(1)} \oplus F_{s^1}^1(T), [m]^{(2)} \oplus F_{s^2}^1(T)\right),$$

where F_k^1 is a PRF keyed by key k. To decrypt a ciphertext $c := \mathsf{Enc}_K^T(m)$, each party P_i sends his sub-key s^i to the decrypter, who uses them to recover the shares of m and reconstruct m.

Double encryption is defined analogously. For keys $K_\alpha = (s_\alpha^1, s_\alpha^2)$ and $K_\beta = (s_\beta^1, s_\beta^2)$, where P_i holds (s_α^i, s_β^i), encryption with tweak T works as follows:

$$\mathsf{Enc}_{K_\alpha,K_\beta}^T(m) = \left([m]^{(1)} \oplus F_{s_\alpha^1}^1(T) \oplus F_{s_\beta^1}^2(T),\ [m]^{(2)} \oplus F_{s_\alpha^2}^1(T) \oplus F_{s_\beta^2}^2(T)\right).$$

Distributed garbling scheme. We now give a high-level description of our two-party distributed garbling scheme $\Pi_{\mathbf{GC}}(P_1, P_2)$; see Figure 2 for the detailed description. As before, for each wire w in the circuit we associate keys $K_{w,0} = (s_{w,0}^1, s_{w,0}^2)$ and $K_{w,1} = (s_{w,1}^1, s_{w,1}^2)$ corresponding to bits '0' and '1', respectively. However, in the distributed setting, each sub-key is only known to one of the two parties; i.e., P_i only knows $(s_{w,0}^i, s_{w,1}^i)$. Each wire is also associated with a mask bit λ_w which is secret shared between the two parties such that no party knows λ_w.

Consider gate G_γ in the circuit with input wires indexed by α and β. As in the non-distributed case, we construct an array containing four rows corresponding to a random permutation of the four possible outcomes of gate G_γ applied to bits b_α and b_β. However, in the distributed case neither party should know what is being encrypted. Recall that in the non-distributed setting, the circuit generator can easily compute $G_\gamma(\lambda_\alpha \oplus b_\alpha, \lambda_\beta \oplus b_\beta)$ to construct the array. However, in the distributed setting, neither party knows (and should *not* know) λ_α or λ_β. Thus, the parties utilize the $\mathcal{F}_{\mathbf{gate}}$ functionality, which takes as input the shares $\langle \lambda_\alpha \rangle \oplus \langle b_\alpha \rangle$ and $\langle \lambda_\beta \rangle \oplus \langle b_\beta \rangle$, and computes a sharing of $G_\gamma(\lambda_\alpha \oplus b_\alpha, \lambda_\beta \oplus b_\beta)$. Let $\langle \sigma_{\gamma,b_\alpha,b_\beta} \rangle = \mathcal{F}_{\mathbf{gate}}^G(\langle b_\alpha \rangle \oplus \langle \lambda_\alpha \rangle, \langle b_\beta \rangle \oplus \langle \lambda_\beta \rangle) \oplus \langle \lambda_\gamma \rangle$. The value $\sigma_{\gamma,b_\alpha,b_\beta}$ denotes which key to encrypt; that is, in row (b_α, b_β) we encrypt key $K_{\gamma,\sigma_{\gamma,b_\alpha,b_\beta}}$. However, we must still enforce that neither party knows what key $K_{\gamma,\sigma_{\gamma,b_\alpha,b_\beta}}$ represents. We handle this by utilizing another functionality, $\mathcal{F}_{\mathbf{oshare}}$. For each of the four $\sigma_{\gamma,b_\alpha,b_\beta}$ values, and for each party P_i, the parties compute $\mathcal{F}_{\mathbf{oshare}}^i(\langle \sigma_{\gamma,b_\alpha,b_\beta} \rangle, s_{\gamma,0}^i, s_{\gamma,1}^i)$. This produces a share of the appropriate sub-key for party P_i, with the crucial fact that P_i does not know which of his sub-keys was shared. The results of $\mathcal{F}_{\mathbf{oshare}}$ are used as the shares to be encrypted.

Note that we can use this two-party distributed garbling scheme as a building block for a somewhat efficient semi-honest two-party secure computation protocol. See Appendix D for the detailed construction. We do not claim that this scheme is superior to existing 2PC protocols; however, it serves as an important building-block to our end goal of an efficient 3PC protocol.

Protocol $\Pi_{\mathbf{GC}}(P_1, P_2)$

Auxiliary Inputs: Security parameter k, circuit $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$.

Parties P_1 and P_2 generate $\langle 1 \rangle \leftarrow \mathcal{F}^1_{\mathbf{const}}$, which they use throughout the protocol.

1. **Generate mask bits:**
   - *Generate masks for P_1's inputs*: For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $\lambda_w \xleftarrow{\$} \{0,1\}$ and computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(\lambda_w)$.
   - *Generate masks for P_2's inputs*: For $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}\}$: P_2 generates $\lambda_w \xleftarrow{\$} \{0,1\}$ and computes $\langle \lambda_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(\lambda_w)$.
   - *Generate masks for inner wires*: For $w \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}-\mathsf{m}\}$: generate $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\mathbf{rand}}$.
   - *Generate masks for output wires*: For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: generate $\langle \lambda_w \rangle \leftarrow \mathcal{F}^0_{\mathbf{const}}$.[a]
2. **Generate sub-keys:**
   - For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}\}$ and $b \in \{0,1\}$: P_i generates sub-keys $s^i_{w,b} \xleftarrow{\$} \{0,1\}^k$.
3. **Construct garbled circuit:**
   - For $\gamma \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}\}$:
     Let $\alpha \leftarrow L(\gamma)$ and $\beta \leftarrow R(\gamma)$ be the indices of the left and right input wires, respectively, of the gate indexed by γ. Compute the following selector bits:

$$\begin{array}{ll} \langle \sigma_{\gamma,0,0} \rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle \lambda_\alpha \rangle, \langle \lambda_\beta \rangle) \oplus \langle \lambda_\gamma \rangle & \langle \sigma_{\gamma,0,1} \rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle \lambda_\alpha \rangle, \langle \lambda_\beta \rangle \oplus \langle 1 \rangle) \oplus \langle \lambda_\gamma \rangle \\ \langle \sigma_{\gamma,1,0} \rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle \lambda_\alpha \rangle \oplus \langle 1 \rangle, \langle \lambda_\beta \rangle) \oplus \langle \lambda_\gamma \rangle & \langle \sigma_{\gamma,1,1} \rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle \lambda_\alpha \rangle \oplus \langle 1 \rangle, \langle \lambda_\beta \rangle \oplus \langle 1 \rangle) \oplus \langle \lambda_\gamma \rangle. \end{array}$$

     Next, compute sharings of the appropriate sub-keys to use for each row:

$$\begin{array}{ll} \left[\hat{s}^1_{\gamma,0,0}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle \sigma_{\gamma,0,0} \rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), & \left[\hat{s}^2_{\gamma,0,0}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle \sigma_{\gamma,0,0} \rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\ \left[\hat{s}^1_{\gamma,0,1}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle \sigma_{\gamma,0,1} \rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), & \left[\hat{s}^2_{\gamma,0,1}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle \sigma_{\gamma,0,1} \rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\ \left[\hat{s}^1_{\gamma,1,0}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle \sigma_{\gamma,1,0} \rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), & \left[\hat{s}^2_{\gamma,1,0}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle \sigma_{\gamma,1,0} \rangle, s^2_{\gamma,0}, s^2_{\gamma,1}) \\ \left[\hat{s}^1_{\gamma,1,1}\right] \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle \sigma_{\gamma,1,1} \rangle, s^1_{\gamma,0}, s^1_{\gamma,1}), & \left[\hat{s}^2_{\gamma,1,1}\right] \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle \sigma_{\gamma,1,1} \rangle, s^2_{\gamma,0}, s^2_{\gamma,1}). \end{array}$$

     Finally, compute the distributed encryptions of the (permuted) sub-keys and selector bits. That is, letting $K_{w,b} = (s^1_{w,b}, s^2_{w,b})$, compute:

$$\begin{array}{l} P[\gamma,0,0] = (P^1[\gamma,0,0], P^2[\gamma,0,0]) \leftarrow \mathsf{Enc}^{\gamma\|0\|0}_{K_{\alpha,0},K_{\beta,0}}(\left[\hat{s}^1_{\gamma,0,0}\right] \| \left[\hat{s}^2_{\gamma,0,0}\right] \| \langle \sigma_{\gamma,0,0} \rangle), \\ P[\gamma,0,1] = (P^1[\gamma,0,1], P^2[\gamma,0,1]) \leftarrow \mathsf{Enc}^{\gamma\|0\|1}_{K_{\alpha,0},K_{\beta,1}}(\left[\hat{s}^1_{\gamma,0,1}\right] \| \left[\hat{s}^2_{\gamma,0,1}\right] \| \langle \sigma_{\gamma,0,1} \rangle), \\ P[\gamma,1,0] = (P^1[\gamma,1,0], P^2[\gamma,1,0]) \leftarrow \mathsf{Enc}^{\gamma\|1\|0}_{K_{\alpha,1},K_{\beta,0}}(\left[\hat{s}^1_{\gamma,1,0}\right] \| \left[\hat{s}^2_{\gamma,1,0}\right] \| \langle \sigma_{\gamma,1,0} \rangle), \\ P[\gamma,1,1] = (P^1[\gamma,1,1], P^2[\gamma,1,1]) \leftarrow \mathsf{Enc}^{\gamma\|1\|1}_{K_{\alpha,1},K_{\beta,1}}(\left[\hat{s}^1_{\gamma,1,1}\right] \| \left[\hat{s}^2_{\gamma,1,1}\right] \| \langle \sigma_{\gamma,1,1} \rangle). \end{array}$$

4. **Output circuit:**
   - Let $GC^i \leftarrow (\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, P^i)$ and let $SK^i \leftarrow \left\{(s^i_{w,0}, s^i_{w,1}) : w \in \{1, \ldots, \mathsf{n}\}\right\}$.
   - P_1 outputs the tuple $\left(GC^1, SK^1, \left\{(\langle b_w \rangle^{(1)}, \langle \lambda_w \rangle^{(1)}, b_w, \lambda_w) : w \in \{1, \ldots, \mathsf{n}_1\}\right\}\right)$.
   - P_2 outputs the tuple $\left(GC^2, SK^2, \left\{(\langle b_w \rangle^{(2)}, \langle \lambda_w \rangle^{(2)}, b_w, \lambda_w) : w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}\}\right\}\right)$.

[a] Note that we do not in fact need to create 'zero' masks for the output wires; we include this step mainly for ease of presentation.

Figure 2: Two-party distributed circuit garbling protocol. For semi-honest security use standard secret sharing for the bits; for malicious security use authenticated secret sharing.

Also note that this distributed garbling scheme can scale to more than two parties, given access to multi-party variants of the necessary functionalities. Thus, we can also achieve (semi-

honest) *multi*-party secure computation using this approach; we leave the development of efficient instantiations of these functionalities as future work.

3.3 Achieving Malicious Security

The semi-honest distributed garbling scheme described in Section 3.2 can be directly adapted to work against a malicious adversary by modifying the hybrid functionalities to work in an authenticated manner; namely, we use authenticated sharings in place of standard secret sharings:

- $\mathcal{F}^1_{\mathbf{const}}()$ and $\mathcal{F}_{\mathbf{rand}}()$: The output share is authenticated.
- $\mathcal{F}^G_{\mathbf{gate}}(\langle a\rangle, \langle b\rangle)$: The inputs and outputs are all authenticated sharings.
- $\mathcal{F}^i_{\mathbf{oshare}}(\langle b\rangle, m_0, m_1)$: The selection bit b is an authenticated sharing.
- $\mathcal{F}^i_{\mathbf{ss}}(b)$: The output is an authenticated sharing of b.

See Appendix A for the detailed descriptions.

We also need to define a notion of *encrypting* authenticated shares. Recall that for an authenticated share $\langle b\rangle = (\langle b\rangle^{(1)}, \langle b\rangle^{(2)})$, we have $\langle b\rangle^{(i)} = (b_i, t_i, k_j)$, where party P_i holds b_i and t_i and party P_j holds k_j. Thus, letting $K = (s^1, s^2)$, we define

$$\mathsf{Enc}^T_K(\langle b\rangle) = (\mathsf{Enc}^1_{s^1,T}(b_1\|t_1\|k_1), \mathsf{Enc}^2_{s^2,T}(b_2\|t_2\|k_2)).$$

On decryption, each party's ciphertext is decrypted and the authenticity of b_1 and b_2 are verified using the (encrypted) tags and keys. Thus, when evaluating a garbled circuit, the party checks the authenticity of the share from the decrypted row of each garbled gate; if the check fails, the party aborts.

Again, we can convert this garbling scheme into a (now *maliciously*-secure) 2PC scheme; see Appendix D for the details. Likewise, we could also construct an MPC variant with efficient *multi-party* instantiations of the underlying functionalities which we leave as future work.

4 Three-Party Computation from Cut-and-choose

As mentioned above, we can directly adapt the distributed garbling scheme to work over multiple parties, and thus construct a 3PC scheme; however, in this case the underlying functionalities need to support multiple parties rather than just two parties and are thus unlikely to be more efficient in practice. Thus, in this section we show how to utilize the maliciously secure two-party distributed garbling scheme from Section 3 to construct a maliciously secure *three*-party secure computation protocol, using almost entirely two-party constructs (the only three-party functionality needed is that of coin-tossing).

We first cover preliminary notions, such as the ideal functionalities we need, in Section 4.1. Then, in Section 4.2 we show how to adapt a combination of two existing cut-and-choose protocols [LP07, LP11] to the three-party setting. Finally, in Section 4.3 we use this "generic" protocol to show how to adapt [Lin13] (the current state-of-the-art Yao-based protocol at the time of writing) to the three-party setting. The cost of each of these three-party protocols is roughly *twice* the computational cost of the underlying two-party protocol they are based on, and roughly *thrice* the communication cost (plus the cost of a small number of OTs per gate, which can be efficiently amortized using OT extension [IKNP03, NNOB12]), and thus we show that we can achieve efficient secure three-party computation at only a small factor of the cost of the most efficient Yao-based two-party protocol.

4.1 Preliminaries

Ideal functionalities. In addition to the ideal functionalities used in the two-party distributed garbling scheme, we need the following additional (maliciously secure) functionalities:

- *Three-party coin-flipping* $\mathcal{F}_{\mathbf{cf}}()$: The functionality outputs a random bit-string $\rho \xleftarrow{\$} \{0,1\}^s$ to each party.
- *One-out-of-two oblivious transfer* $\mathcal{F}_{\mathbf{ot}}^{i,j}(b, m_0, m_1)$: The functionality takes as input a choice bit b from party P_i and messages m_0, m_1 from P_j, and outputs m_b to party P_i.
- *ZKPoK of extended Diffie-Hellman tuple* $\mathcal{F}_{\mathbf{zkpok}}^{i,j}(a, (g, h_0, h_1, \{u_i, v_i\}_i))$: The functionality takes as input a from party P_i, and tuple $(g, h_0, h_1, \{u_i, v_i\}_i)$ from party P_j, and outputs 1 to party P_j if either all tuples in $\{(g, h_0, u_i, v_i)\}_i$ are Diffie-Hellman tuples with $h_0 = g^a$ or all tuples in $\{(g, h_1, u_i, v_i)\}_i$ are Diffie-Hellman tuples with $h_1 = g^a$, and 0 otherwise.

These can all be efficiently instantiated in a standard fashion. We can implement $\mathcal{F}_{\mathbf{cf}}$ in the Random Oracle Model using three commitments and openings. The $\mathcal{F}_{\mathbf{ot}}$ functionality can be instantiated using any maliciously secure OT implementation, such as [PVW08]. Finally, $\mathcal{F}_{\mathbf{zkpok}}$ can be efficiently instantiated using the protocol in [LP11, Section B].

Distributed garbled circuits for three parties. Note that the garbling protocol $\Pi_{\mathbf{GC}}$ in Figure 2 only garbles a circuit containing inputs from two parties. We can easily adapt this to support input from a third (external) party as follows. Let $\Pi'_{\mathbf{GC}}(P_1, P_2)$ be the same as $\Pi_{\mathbf{GC}}(P_1, P_2)$ except for the following modifications:

- All of the operations over P_2's input now operate over wires $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$.
- In Step 1, we add the following sub-step for generating shares for P_3's input wires:
  - For $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$: generate $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\mathbf{rand}}$.
- In Step 4, party P_i outputs $\left\{\langle \lambda_w \rangle^{(i)} : w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}\right\}$ in addition to his normal outputs.

4.2 Achieving Malicious Security for Three Parties

Note that our two-party distributed garbling scheme has the property that if at most one of the two parties is corrupt, the garbling of circuit C either correctly evaluates C on P_1's and P_2's inputs, or causes the evaluator to abort. That is, a malicious party cannot "alter" the garbling to evaluate some circuit other than C. Now, if both P_1 and P_2 are corrupt, they can of course garble an arbitrary circuit. This suggests the following approach to three-party computation: If either P_1 or P_2 are honest, we need only construct a single garbled circuit, which is sent to P_3 to be evaluated. To cover the case where both P_1 and P_2 are corrupt, we use cut-and-choose to prevent P_3 from evaluating a maliciously constructed circuit. In what follows, we utilize existing cut-and-choose protocols from the literature, in particular a combination of [LP07] and [LP11], and "plug in" our distributed garbling scheme as necessary. Thus, security mostly follows from the security proofs of the underlying cut-and-choose protocols. In Section 4.3, we show how we can use this protocol in an adaptation of Lindell's protocol [Lin13] to the three-party setting.

The basic intuition for security is as follows. Cut-and-choose is used to prevent P_3 from evaluating maliciously constructed circuits when both P_1 and P_2 are malicious. For the case where either P_1 or P_2 is honest, $\Pi'_{\mathbf{GC}}(P_1, P_2)$ assures us that the garbled circuit constructed between P_1 and P_2 is either correctly constructed or causes P_3 to abort (independent of any party's input).

Protocol description. We assume the reader is familiar with the cut-and-choose technique; here we briefly discuss the main technical challenges that result from a naïve application of cut-and-choose and how we address them.

- *Input Inconsistency.* The use of cut-and-choose produces multiple garbled circuits to be evaluated by P_3. The idea with this attack is that a given party (either P_1 or P_2 in the three-party case) can give inconsistent sub-keys in each of these circuits such that P_3 ends up evaluating different inputs for P_1/P_2 instead of consistent inputs across all garbled circuits. This is a well-known attack, and there are multiple solutions in the two-party setting. Here, we use the Diffie-Hellman pseudorandom synthesizer trick [MF06, LP11] and adapt it in a straightforward manner to the three-party setting.
- *Selective Failure.* This attack arises any time the parties execute an OT to send the sub-keys for P_3's input. Note that if the sender in the OT (either party P_1 or P_2) inputs one valid label and one invalid label, he can learn a bit of P_3's input by learning whether the garbled circuit evaluation fails or not. We circumvent this problem by directly applying the "XOR-tree" approach [LP07, Woo07].

We now give a high-level description of our protocol.

1. The parties first replace the input circuit C^0 with a circuit C, where the only difference is each of P_3's input wires is replaced by an XOR of s new input wires, preventing either party P_1 or P_2 from launching a selective failure attack on P_3's input choices.

2. P_1 and P_2 generate the required commitments needed for input consistency, as is done in [LP11].

3. P_1 and P_2 construct s garbled circuits using $\Pi'_{\mathbf{GC}}$ and the input sub-keys generated as in [LP11].

4. P_1 and P_2 compute authenticated sharings (between each other; P_3 is not involved here) of their input bits.

5. P_1 and P_2 both run (separately) an OT protocol with P_3 for each of P_3's input wires, where P_1/P_2 input their sub-keys and P_3 chooses based on his input. (Note that any cheating by P_1/P_2 here will be caught with high-probability by the cut-and-choose step below.) Thus, P_3 now has keys for each of his input bits.

6. P_1 and P_2 send the (distributed) garbled circuits, along with the input consistency commitments, to P_3.

7. All three parties run a coin-tossing protocol to determine which circuits for P_3 to open and which to evaluate.

8. For the evaluation circuits, P_1 and P_2 send the sub-keys and selector bits for their inputs to P_3. Note that we need to be careful in this step, as we need to enforce that, for example, P_1 uses the same input as was shared in Step 2 above. This is accomplished as follows. Recall that P_1 and P_2 have sharings of each other's inputs and mask bits, all of which are authenticated. Thus, P_1 can send the (authenticated) share of her masked input to P_2, who can verify its authenticity, and thus reconstruct the masked input bit using his own share. This allows an honest P_2 to send the correct sub-key (correct in the sense that it corresponds to P_1's input shared in Step 2) to P_3, even with a malicious P_1.

9. For the check circuits, P_1 and P_2 send the required information for P_3 to decrypt the check circuits and verify correctness. If any of these check circuits are incorrectly constructed, P_3 aborts; otherwise, he has high confidence that the majority of the evaluation circuits are correctly constructed.

10. For the evaluation circuits, P_3 checks for input consistency against the sub-keys sent by P_1 and P_2 in Step 8 using the zero-knowledge proof-of-knowledge protocol shown in [LP11], aborting on any inconsistency.

11. Finally, P_3 evaluates the evaluation circuits, outputting the majority over the circuits' output.

See below for the full protocol description.

Protocol $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$

Auxiliary Inputs: Security parameter k, statistical security parameter s, circuit C^0, cyclic group $\mathbb{G}$ with (prime) order q and generator g, and randomness extractor H.

Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}_2\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_2+1, \ldots, \mathsf{n}\}$, P_3 has inputs b_w.

1. Each party replaces C^0 with a circuit C where each of P_3's input wires is replaced by an exclusive-or of s new input wires. We let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$, and denote P_3's new inputs by $\hat{b}_w$.
2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $a^1_{w,0}, a^1_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^1_{w,0}}), (w, 1, g^{a^1_{w,1}})\right\}$.
   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 generates $a^2_{w,0}, a^2_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^2_{w,0}}), (w, 1, g^{a^2_{w,1}})\right\}$.
   For $j \in \{1, \ldots, s\}$: P_i, for $i \in \{1, 2\}$, generates $r^i_j \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(j, g^{r^i_j})\right\}$.
   For $j \in \{1, \ldots, s\}$: P_1 and P_2 run up to Step 2 ("Generate sub-keys") of $\Pi^3_{\mathbf{GC}}(P_1, P_2)$, where the parties do the following in the jth iteration:
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates sub-keys $s^1_{w,b\oplus\lambda_{w,j},j} \leftarrow H(g^{a^1_{w,b} \cdot r^1_j})$ for $b \in \{0, 1\}$.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 generates sub-keys $s^2_{w,b\oplus\lambda_{w,j},j} \leftarrow H(g^{a^2_{w,b} \cdot r^2_j})$ for $b \in \{0, 1\}$.
   - All other sub-keys are generated in the normal fashion.
3. For $j \in \{1, \ldots, s\}$: P_1 and P_2 continue their executions of $\Pi^3_{\mathbf{GC}}(P_1, P_2)$, producing garbled circuit GC_j.
4. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.
5. For $j \in \{1, \ldots, s\}$ and $w \in \{\mathsf{n}_2+1, \ldots, \mathsf{n}\}$: P_1 and P_2 exchange $\langle \lambda_{w,j} \rangle$ with each other, reconstructing $\lambda_{w,j}$ locally. Both P_1 and P_2 send $\lambda_{w,j}$ to P_3.
   For $w \in \{\mathsf{n}_2 + 1, \ldots, \mathsf{n}\}$: P_i, for $i \in \{1, 2\}$, and P_3 run $\mathcal{F}_{\mathbf{ot}}$, with P_i as the sender inputting $\left(\left\{s^i_{w,\lambda_{w,j},j}\right\}_{j\in\{1,\ldots,s\}}, \left\{s^i_{w,\lambda_{w,j}\oplus 1,j}\right\}_{j\in\{1,\ldots,s\}}\right)$ and P_3 as the receiver inputting $\hat{b}_w$.
6. P_i, for $i \in \{1, 2\}$, sends the sets constructed in Step 2, along with the garbled circuit $\left\{GC^i_j\right\}^s_{i=1}$, to P_3.
7. The parties compute $\rho \leftarrow \mathcal{F}_{\mathbf{cf}}$. Let $\mathcal{CC} = \{i : \rho_i = 1\}$, and let $\mathcal{EC} = \{1, \ldots, s\} \setminus \mathcal{CC}$.
8. For $j \in \mathcal{EC}$:
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $\langle b_w \rangle^{(1)} \oplus \langle \lambda_{w,j} \rangle^{(1)}$ to P_2, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$: P_2 sends $\langle b_w \rangle^{(2)} \oplus \langle \lambda_{w,j} \rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
9. For $j \in \mathcal{CC}$:
   - P_i, for $i \in \{1, 2\}$, does the following:
     - Sends r^i_j to P_3, and P_3 checks that these values are consistent with the pairs $\left\{(j, g^{r^i_j})\right\}$ sent before.

- For $w \in \{1, \ldots, \mathsf{n}\}$: Sends sub-keys $s^i_{w,0,j}$ and $s^i_{w,1,j}$, mask bit share $\lambda^{(i)}_{w,j}$, and the keys to the authenticated bits to P_3.

- Given the above information, P_3 reconstructs all input labels and verifies they match with those labels sent previously. Also, using said labels, P_3 verifies that the garbled circuit is correctly constructed.

10. For $j \in \mathcal{EC}$:
    - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $g^{a^1_{w,b_w} \cdot r^1_j}$ to P_3, who computes $s^1_{w,b_w \oplus \lambda_{w,j},j} \leftarrow H(g^{a^1_{w,b_w} \cdot r^1_j})$.
    - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 sends $g^{a^2_{w,b_w} \cdot r^2_j}$ to P_3, who computes $s^2_{w,b_w \oplus \lambda_{w,j},j} \leftarrow H(g^{a^2_{w,b_w} \cdot r^2_j})$.

    For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_3 run $\mathcal{F}_{\mathbf{zkpok}}$, with P_1 as the prover inputting a^1_{w,b_w} and P_3 as the verifier inputting $\left(g, g^{a^1_{w,0}}, g^{a^1_{w,1}}, \left\{(g^{r^1_j}, g^{a^1_{w,b_w} \cdot r^1_j})\right\}_{j \in \mathcal{EC}}\right)$.

    For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 and P_3 run $\mathcal{F}_{\mathbf{zkpok}}$, with P_2 as the prover inputting a^2_{w,b_w} and P_3 as the verifier inputting $\left(g, g^{a^2_{w,0}}, g^{a^2_{w,1}}, \left\{(g^{r^2_j}, g^{a^2_{w,b_w} \cdot r^2_j})\right\}_{j \in \mathcal{EC}}\right)$.

11. For $j \in \mathcal{EC}$: P_3 evaluates circuit GC_j using $\left\{(s^1_{w,b_w \oplus \lambda_{w,j},j}, s^2_{w,b_w \oplus \lambda_{w,j},j}, b_w \oplus \lambda_{w,j})\right\}_{w \in \{1,\ldots,\mathsf{n}\}}$ as inputs. P_3 outputs the majority output over the evaluated circuits.

Theorem 1. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof. See Appendix E. □

4.3 Adapting [Lin13] to the Three-party Setting

The 3PC protocol described above has a replication factor of roughly 3×; namely, for statistical security parameter s, the actual probability of cheating is roughly $2^{-0.32s}$ [LP11]. Thus, for a desired error probability of 2^{-40} a total of 128 circuits need to be garbled. Recently, Lindell [Lin13] showed a beautiful construction which removes this replication factor in the two-party setting; that is, for a cheating probability of 2^{-s} the sender needs to garble *only* s circuits. In this section we show how to adapt this protocol to the three-party setting.

Lindell's construction works in two phases. In the first phase, the parties do a standard cut-and-choose, with P_1 constructing s circuits (for error probability 2^{-s}) and P_2 opening half of them. If, during evaluation, P_2 finds that two or more circuits have conflicting outputs, it stores these conflicting outputs as a "proof-of-cheating" ϕ. In the second phase, the parties run a circuit which takes as input from P_1 her original input x, and from P_2 the "proof-of-cheating" ϕ. If ϕ is a valid proof, then the circuit reveals x to P_2, who can then compute the output himself; otherwise P_2 gets no output. Thus, this second phase enforces that if P_1 cheated in the cut-and-choose then P_2 learns P_1's input. See [Lin13] for further details.

To adapt this to the three-party setting, we proceed as follows. For the circuit in the first phase, we essentially just run $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$, with the same tweaks as are used in [Lin13] (namely, the use of encoded output translation tables and doing circuit evaluation before circuit checking).

For the second phase circuit, we run into some issues, due to Lindell's scheme being inherently a "two-party" approach. Recall that this circuit is constructed in such a way that if P_2 receives any conflicting outputs when evaluating, he inputs these outputs as a "proof-of-cheating" in order

to reveal P_1's input. At first glance, it appears this technique would *not* work in the three-party setting because in that case P_3 needs to learn *both* P_1's and P_2's inputs to reconstruct the output; however, it could be the case that only *one* of these two parties is cheating. Recall, however, that our distributed garbling scheme enforces that as long as *one* of the two parties is honest, the garbled circuits are "correct" in the sense that they either correctly compute the desired circuit are cause a failure independent of any party's input. Thus, P_3 only finds mismatched outputs in the case where *both* P_1 and P_2 cheat, making it okay at this point to reveal both those parties' inputs in the second phase circuit.

Another issue arises in how this circuit is constructed. In Lindell's two-party scheme, P_1 hardwires the output keys into the circuit. In a naïve adaptation to the three-party setting, both P_1 and P_2 would need to hardwire their output *sub*-keys into the circuit. However, this would allow each party to learn the others' sub-keys for the output, which leads to the following attack by a colluding P_1 and P_3: During the construction of the second phase circuit, P_1 learns P_2's output sub-keys, and he sends these, as well as his own output sub-keys, to P_3. Now, when P_3 evaluates the circuit, he can input conflicting outputs as his "proof-of-cheating" because he knows all of the outputs keys, thus allowing P_1 and P_3 to learn P_2's input. We can fix this by having the output sub-keys of P_1 and P_2 be inputs to the circuit, rather than hardcoded. However, this raises another issue, as P_3 cannot verify that the sub-keys input by P_1 and P_2 are the correct ones. Thus, we modify the circuit to output these sub-keys in the clear, allowing P_3 to do this check.

See below for the full protocol description.

Protocol $\Pi^{m-lindell}_{\mathbf{3PC}}(P_1, P_2, P_3)$

Auxiliary Inputs: Security parameter k, statistical security parameter s, circuit C^0, cyclic group $\mathbb{G}$ with (prime) order q and generator g, and randomness extractor H.

Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1+1, \ldots, \mathsf{n}_2\}$, P_2 has inputs b_w; for $w \in \{\mathsf{n}_2+1, \ldots, \mathsf{n}\}$, P_3 has inputs b_w.

1. Each party replaces C^0 with a circuit C where each of P_3's input wires is replaced by an exclusive-or of s new input wires. We let $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$, and denote P_3's new inputs by $\hat{b}_w$.
2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates $a^1_{w,0}, a^1_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^1_{w,0}}), (w, 1, g^{a^1_{w,1}})\right\}$.
   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 generates $a^2_{w,0}, a^2_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(w, 0, g^{a^2_{w,0}}), (w, 1, g^{a^2_{w,1}})\right\}$.
   For $w \in \{\mathsf{n} + \mathsf{q} - \mathsf{m} + 1, \ldots, \mathsf{n} + \mathsf{q}\}$: P_i, for $i \in \{1, 2\}$, generates $o^i_{w,0}, o^i_{w,1} \xleftarrow{\$} \{0,1\}^k$.
   For $j \in \{1, \ldots, s\}$: P_i, for $i \in \{1, 2\}$, generates $r^i_j \xleftarrow{\$} \mathbb{Z}_q$ and constructs set $\left\{(j, g^{r^i_j})\right\}$.
   For $j \in \{1, \ldots, s\}$: P_1 and P_2 run up to Step 2 ("Generate sub-keys") of $\Pi'_{\mathbf{GC}}(P_1, P_2)$, where the parties do the following:
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 generates sub-keys $s^1_{w,b\oplus\lambda_{w,j},j} \leftarrow H(g^{a^1_{w,b} \cdot r^1_j})$ for $b \in \{0,1\}$.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 generates sub-keys $s^2_{w,b\oplus\lambda_{w,j},j} \leftarrow H(g^{a^2_{w,b} \cdot r^2_j})$ for $b \in \{0,1\}$.
   - For $w \in \{\mathsf{n} + \mathsf{q} - \mathsf{m} + 1, \ldots, \mathsf{n} + \mathsf{q}\}$: P_i sets $s^i_{w,b\oplus\lambda_{w,j}} \leftarrow o^i_{w,b}$.
   - All other sub-keys are generated in the normal fashion.
3. For $j \in \{1, \ldots, s\}$: P_1 and P_2 continue their executions of $\Pi'_{\mathbf{GC}}(P_1, P_2)$, producing (distributed) garbled circuit $GC_j := (GC^1_j, GC^2_j)$.
4. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_1 and P_2 compute $\langle b_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.

5. For $j \in \{1,\ldots,s\}$ and $w \in \{\mathsf{n}_2+1,\ldots,\mathsf{n}\}$: P_1 and P_2 exchange $\langle \lambda_{w,j} \rangle$ with each other, reconstructing $\lambda_{w,j}$ locally. Both P_1 and P_2 send $\lambda_{w,j}$ to P_3.
   For $w \in \{\mathsf{n}_2+1,\ldots,\mathsf{n}\}$: P_i, for $i \in \{1,2\}$, and P_3 run $\mathcal{F}_{\mathbf{ot}}$, with P_i as the sender inputting $\left(\left\{s^i_{w,\lambda_{w,j},j}\right\}_{j\in\{1,\ldots,s\}}, \left\{s^i_{w,\lambda_{w,j}\oplus 1,j}\right\}_{j\in\{1,\ldots,s\}}\right)$ and P_3 as the receiver inputting $\hat{b}_w$.
6. For $i \in \{1,2\}$: P_i sends the sets constructed in Step 2, along with the garbled circuit $\left\{GC^i_j\right\}_{j=1}^{s}$, to P_3. In addition, P_i sends the *encoded output translation table* $\left\{(H(o^i_{w,0}), H(o^i_{w,1}))\right\}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}^{\mathsf{n}+\mathsf{q}}$ to P_3.
7. The parties compute $\rho \leftarrow \mathcal{F}_{\mathbf{cf}}$. Let $\mathcal{CC} = \{i : \rho_i = 1\}$, and let $\mathcal{EC} = \{1,\ldots,s\} \setminus \mathcal{CC}$.
8. For $j \in \mathcal{EC}$ (the *evaluation* circuits):
   - For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 sends $\langle b_w \rangle^{(1)} \oplus \langle \lambda_{w,j} \rangle^{(1)}$ to P_2, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}\}$: P_2 sends $\langle b_w \rangle^{(2)} \oplus \langle \lambda_{w,j} \rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_{w,j}$ locally. P_1 sends $(s^1_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3, and P_2 sends $(s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})$ to P_3.
   - For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 sends $k^1_{w,j} := g^{a^1_{w,b_w}\cdot r^1_j}$ to P_3, who computes $s^1_{w,b_w\oplus\lambda_{w,j},j} \leftarrow H(g^{a^1_{w,b_w}\cdot r^1_j})$.
   - For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}_2\}$: P_2 sends $k^2_{w,j} := g^{a^2_{w,b_w}\cdot r^2_j}$ to P_3, who computes $s^2_{w,b_w\oplus\lambda_{w,j},j} \leftarrow H(g^{a^2_{w,b_w}\cdot r^2_j})$.
   - P_3 evaluates circuit GC_j using $\left\{(s^1_{w,b_w\oplus\lambda_{w,j},j}, s^2_{w,b_w\oplus\lambda_{w,j},j}, b_w \oplus \lambda_{w,j})\right\}_{w\in\{1,\ldots,\mathsf{n}\}}$ as inputs.

   P_3 uses the encoded output translation tables sent in Step 6 to check if he received exactly one valid output value for each output wire. If not, he stores these outputs as o^j_0 and o^j_1 and continues.
9. P_1 and P_2 construct a circuit C' as follows:
   - P_1 inputs string $x \in \{0,1\}^{\mathsf{n}_1}$ and strings $o^i_{w,0}, o^i_{w,1} \in \{0,1\}^k$, for $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1,\ldots,\mathsf{n}+\mathsf{q}\}$.
   - P_1 inputs string $y \in \{0,1\}^{\mathsf{n}_2-\mathsf{n}_1}$ and strings $o^2_{w,0}, o^2_{w,1} \in \{0,1\}^k$, for $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1,\ldots,\mathsf{n}+\mathsf{q}\}$.
   - P_3 inputs $o_0, o_1 \in \{0,1\}^k$.
   - If there exists some j such that $o^1_{j,0}\|o^2_{j,0} = o^j_0$ and $o^1_{j,1}\|o^2_{j,1} = o^j_1$, then P_3's output is $x\|y$; otherwise P_3 receives no output.
   - The circuit also outputs the values $\left\{o^1_{w,0}, o^1_{w,1}, o^2_{w,0}, o^2_{w,1}\right\}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}^{\mathsf{n}+\mathsf{q}}$ input by parties P_1 and P_2 above.

   The parties run $\Pi^m_{\mathbf{3PC}}(P_1,P_2,P_3)$ on circuit C' as follows:
   - P_1 inputs her input $x = b_1 \ldots b_{\mathsf{n}_1}$; P_2 inputs his input $y = b_{\mathsf{n}_1+1} \ldots b_{\mathsf{n}_2}$.
   - If P_3 received two conflicting outputs $o^1_0\|o^2_0$ and $o^1_1\|o^2_1$ for some circuit $j \in \{1,\ldots,s\}$ in Step 8, then he inputs these values; otherwise he inputs garbage.
   - The garbled circuit uses the same $a^1_{w,0}, a^1_{w,1}, a^2_{w,0}, a^2_{w,1}$ values as in Step 2.

   P_3 verifies that the values $\left\{o^1_{w,0}, o^1_{w,1}, o^2_{w,0}, o^2_{w,1}\right\}_{w=\mathsf{n}+\mathsf{q}-\mathsf{m}+1}^{\mathsf{n}+\mathsf{q}}$ output by C' match those in the encoded output translation tables sent in Step 6
10. For $j \in \mathcal{CC}$ (the *check* circuits):
    - P_i, for $i \in \{1,2\}$, does the following:
      - Sends r^i_j to P_3, and P_3 checks that these values are consistent with the pairs $\left\{(j, g^{r^i_j})\right\}$ sent before.
      - For $w \in \{1,\ldots,\mathsf{n}\}$: Sends sub-keys $s^i_{w,0,j}$ and $s^i_{w,1,j}$, mask bit share $\lambda^{(i)}_{w,j}$, and the keys to the authenticated bits to P_3.
    - Given the above information, P_3 reconstructs all input labels and verifies they match with those labels sent previously. Also, using said labels, P_3 verifies that the garbled circuit is correctly constructed.
11. For the cut-and-choose computation from Step 9, let $\widehat{\mathcal{EE}}$ be the check circuits, let $\widehat{r}^i_j$ be analogous to the r^i_j values from Step 2, and let $\widehat{k}^i_{w,j}$ be analogous to the $k^i_{w,j}$ from Step 6.
    For $w \in \{1,\ldots,\mathsf{n}_1\}$: P_1 and P_3 run a zero-knowledge proof-of-knowledge, with P_1 proving that there exists some $b_w \in \{0,1\}$ such that for every $j \in \mathcal{EE}$ and for every $j' \in \widehat{\mathcal{EE}}$, $k^1_{w,j} = g^{a^1_{w,b_w}\cdot r^1_j}$ and $\widehat{k}^1_{w,j'} = g^{a^1_{w,b_w}\cdot \widehat{r}^1_{j'}}$.
    For $w \in \{\mathsf{n}_1+1,\ldots,\mathsf{n}_2\}$: P_2 and P_3 run a zero-knowledge proof-of-knowledge, with P_2 proving that there exists some $b_w \in \{0,1\}$ such that for every $j \in \mathcal{EE}$ and for every $j' \in \widehat{\mathcal{EE}}$, $k^2_{w,j} = g^{a^2_{w,b_w}\cdot r^2_j}$ and $\widehat{k}^2_{w,j'} = g^{a^2_{w,b_w}\cdot \widehat{r}^2_{j'}}$.

12. P_3 either outputs the output received in the evaluation circuits, or, if P_3 received any inconsistent inputs in Step 8, then it locally computes $f(x, y, z)$, where x and y are the inputs P_3 received in Step 9, and z is P_3's own input.

Theorem 2. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi_{\mathbf{3PC}}^{m-lindell}(P_1, P_2, P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof. See Appendix F. □

4.4 Efficiency

We now argue why our 3PC protocol is roughly twice as expensive in terms of computation as the underlying 2PC protocol we utilize, and roughly three times as expensive in terms of communication. Both protocols are very similar to the underlying 2PC protocol they are based on; the major change in terms of computation cost is that P_3 needs to do twice the work (due to needing to communicate with *both* P_1 and P_2) as compared to the evaluator in the underlying 2PC protocol. Finally, the cost for P_1 and P_2 to distributively garble a circuit is a small number of OTs per gate, and this can be amortized using OT extension techniques [IKNP03].

In terms of communication cost, both P_1 and P_2 need to send their half of the distributed garbled circuit to P_3, and the communication cost of actually constructing a distributed garbled circuit is roughly the cost of a standard garbled circuit. Thus, the overall communication is about three times that of the underlying 2PC protocol.

5 Efficient Instantiations of Necessary Functionalities

In this section, we show how to efficiently instantiate the calls to the two-party functionalities used in our protocols.

Semi-honest setting. The hybrids for our semi-honest protocol can be instantiated in the standard fashion, see Appendix A. For instantiating $\mathcal{F}_{\mathbf{oshare}}$, we need a single call to $\mathcal{F}_{\mathbf{ot}}$, but otherwise the protocol is fairly straight-forward; see Appendix B.

Malicious setting. For the malicious setting we use ideas from the NNOB protocol [NNOB12]. The key notion is that of an (obliviously) *authenticated bit*, or *aBit* for short. Namely, for a bit b authenticated towards P_i, P_i holds both the bit b and a mask M_b, with P_j holding the authentication key K_b and a global key Δ_j such that $M_b = K_b \oplus b\Delta_j$. In [NNOB12] they show how to efficiently construct such authenticated bits in the Random Oracle model using OT extension [IKNP03]. Besides the one-time cost of using OT extension, Nielsen et al. [NNOB12] claim a protocol requiring only 8 calls to the underlying hash function H per aBit.[3]

In the following, we briefly describe how to instantiate our hybrid functionalities assuming sufficiently many aBits. We do not discuss $\mathcal{F}_{\mathbf{cf}}$, $\mathcal{F}_{\mathbf{ot}}$, and $\mathcal{F}_{\mathbf{zkpok}}$ here because they can be instantiated without using aBits.

[3]The protocol described in [NNOB12] uses 59 calls to H.

- $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{ss}}$, $\mathcal{F}_{\mathbf{rand}}$, $\mathcal{F}^{G}_{\mathbf{gate}}$. These hybrids can be directly instantiated using the protocols described in [NNOB12] (we assume when using $\mathcal{F}_{\mathbf{gate}}$ that only AND and XOR gates are used). We now describe the cost of these protocols in terms of the number of calls to the underlying hash function H: $\mathcal{F}_{\mathbf{const}}$ requires no calls to H; $\mathcal{F}_{\mathbf{ss}}$ requires one aBit and thus 8 calls to H; $\mathcal{F}_{\mathbf{rand}}$ requires two aBits and thus 16 calls to H; $\mathcal{F}^{XOR}_{\mathbf{gate}}$ can be computed locally; and $\mathcal{F}^{AND}_{\mathbf{gate}}$ requires 584 calls to H.
- $\mathcal{F}_{\mathbf{oshare}}$. The implementation of $\mathcal{F}_{\mathbf{oshare}}$ requires calls to an appropriate underlying OT primitive. To ensure that the sharings are shuffled according to the authenticated bit, we introduce a primitive called *receiver-authenticated string oblivious transfer*, or in short, RaOT. As the name suggests, RaOT receives two input strings $s_0, s_1 \in \{0,1\}^*$ from the sender P_j, and an aBit $\langle b \rangle^{(i)}$ from the receiver P_i, and outputs s_b to the receiver (i.e., the aBit commits the receiver to his selection).
  A naïve implementation of RaOT can be obtained by using the authenticated OT primitive from NNOB for each bit of the strings s_0 and s_1. However, this is a clear overkill as we only need the bit of the receiver to be authenticated. We have a more efficient implementation of RaOT which requires one aBit plus four additional calls to H, see Appendix C. $\mathcal{F}_{\mathbf{oshare}}$ can be instantiated using two calls to RaOT and a single aBit; again, see Appendix C. Thus, $\mathcal{F}_{\mathbf{oshare}}$ requires a total of 32 calls to H.

Acknowledgments

Research of the second author was supported in part by NSF award #1111599 and by the US Army Research Laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. Research of the third author was conducted with Government support through the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the US Government, the UK Ministry of Defense, or the UK Government.

References

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *13th Intl. Conference on Financial Cryptography and Data Security*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer-Verlag, 2009.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *15th ACM Conference on Computer and Communications Security*, pages 257–266, Alexandria, Virginia, USA, October 27–31, 2008. ACM Press.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor,

Advances in Cryptology—Eurocrypt 2011, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer-Verlag. Full version available at `https://eprint.iacr.org/2010/514`.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *19th ACM Conference on Computer and Communications Security*, pages 784–796. ACM Press, October 16–18, 2012. Full version available at `https://eprint.iacr.org/2012/265`.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *13th European Symposium on Research in Computer Security (ESORICS)*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer-Verlag, 2008. Full version available at `https://eprint.iacr.org/2008/289`.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, Maryland, USA, May 14–16, 1990. ACM Press.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In Ian Goldberg, editor, *19th USENIX Security Symposium*, Washington, D.C., USA, August 11–13, 2010. USENIX Association.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. In Angelos D. Keromytis, editor, *16th Intl. Conference on Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer-Verlag, 2012. Full version available at `https://eprint.iacr.org/2011/662`.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000. Full version available at `https://eprint.iacr.org/1998/018`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, USA, October 14–17, 2001. IEEE Computer Society Press. Full version available at `https://eprint.iacr.org/2000/067`.

[CHK+12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *Topics in Cryptology—CT-RSA 2012*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer-Verlag, 2012.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages

160–179, Irvine, CA, USA, March 18–20, 2009. Springer-Verlag. Full version available at https://eprint.iacr.org/2008/415.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology—Crypto 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394, Santa Barbara, CA, USA, August 14–18, 2005. Springer-Verlag. Full version available at https://eprint.iacr.org/2005/262.

[DKL+12] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *8th International Conference on Security in Communication Networks*, volume 7485 of *Lecture Notes in Computer Science*, pages 241–263, Amalfi, Italy, September 5–7, 2012. Springer-Verlag. Full version available at https://eprint.iacr.org/2012/262.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority, or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *18th European Symposium on Research in Computer Security (ESORICS)*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2013. Full version available at https://eprint.iacr.org/2012/642.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology—Crypto 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer-Verlag. Full version available at https://eprint.iacr.org/2011/535.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In Amit Sahai, editor, *10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641, Tokyo, Japan, March 3–6, 2013. Springer-Verlag. Full version available at https://eprint.iacr.org/2012/512.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City,, New York, USA, May 25–27, 1987. ACM Press.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge University Press, 2009.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *ISOC Network and Distributed System Security Symposium—NDSS 2012*, San Diego, California, USA, February 5–8, 2012. The Internet Society.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In David Wagner, editor, *20th USENIX Security Symposium*, San Francisco, California, USA, August 8–12, 2011. USENIX Association.

[HKE13] Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology—Crypto 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 18–35, Santa Barbara, CA, USA, August 18–22, 2013. Springer-Verlag. Full version available at `https://eprint.iacr.org/2013/081`.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *17th ACM Conference on Computer and Communications Security*, pages 451–462, Chicago, Illinois, USA, October 4–8, 2010. ACM Press. Full version available at `https://eprint.iacr.org/2010/365`.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology—Crypto 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161, Santa Barbara, CA, USA, August 17–21, 2003. Springer-Verlag.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology—Crypto 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer-Verlag.

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Towards billion-gate secure computation with malicious adversaries. In Tadayoshi Kohno, editor, *21st USENIX Security Symposium*, Bellevue, Washington, USA, August 8–10, 2012. USENIX Association. Full version available at `https://eprint.iacr.org/2012/179`.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *20th ACM Conference on Computer and Communications Security*, pages 549–560, Berlin, Germany, November 4–8, 2013. ACM Press. Full version available at `https://eprint.iacr.org/2013/143`.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology—Crypto 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 1–17, Santa Barbara, CA, USA, August 18–22, 2013. Springer-Verlag.

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *Advances in Cryptology—Crypto 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 259–276, Santa Barbara, CA, USA, August 14–18, 2011. Springer-Verlag. Full version available at `https://eprint.iacr.org/2011/435`.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology—Eurocrypt 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78, Barcelona, Spain, May 20–24, 2007. Springer-Verlag.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346, Providence, RI, USA, March 28–30, 2011. Springer-Verlag. Full version available at `https://eprint.iacr.org/2010/284`.

[LPS08] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *6th International Conference on Security in Communication Networks*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20, Amalfi, Italy, September 10–12, 2008. Springer-Verlag.

[MF06] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473, New York, NY, USA, April 24–26, 2006. Springer-Verlag.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology—Crypto 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53, Santa Barbara, CA, USA, August 18–22, 2013. Springer-Verlag. Full version available at `https://eprint.iacr.org/2013/051`.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology—Crypto 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer-Verlag. Full version available at `https://eprint.iacr.org/2011/091`.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer-Verlag, March 15–17, 2009. Full version available at `https://eprint.iacr.org/2008/427`.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267, Tokyo, Japan, December 6–10, 2009. Springer-Verlag. Full version available at `https://eprint.iacr.org/2009/314`.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *Advances in Cryptology—Crypto 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571, Santa Barbara, CA, USA, August 17–21, 2008. Springer-Verlag. Full version available at `https://eprint.iacr.org/2007/348`.

[sS11] abhi shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *Advances in Cryptology—Eurocrypt 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405, Tallinn, Estonia, May 15–19, 2011. Springer-Verlag. Full version available at `https://eprint.iacr.org/2011/533`.

[sS13] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *20th ACM Conference on Computer and Communications Security*, pages 523–534, Berlin, Germany, November 4–8, 2013. ACM Press. Full version available at `https://eprint.iacr.org/2013/196`.

[SZ13] Thomas Schneider and Michael Zohner. GMW vs. Yao? efficient secure two-party computation with low depth circuits. In Ahmad-Reza Sadeghi, editor, *17th Intl. Conference on Financial Cryptography and Data Security*, volume 7859 of *Lecture Notes in Computer Science*, pages 275–292, Okinawa, Japan, April 1–5, 2013. Springer-Verlag.

[Woo07] David P. Woodruff. Revisiting the efficiency of malicious two-party computation. In Moni Naor, editor, *Advances in Cryptology—Eurocrypt 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 79–96, Barcelona, Spain, May 20–24, 2007. Springer-Verlag. Full version available at `https://eprint.iacr.org/2006/397`.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

Appendices

A Hybrid Functionalities

Secret sharing of a constant bit. The functionality $\mathcal{F}^b_{\mathbf{const}}$ is parameterized by a bit b, and outputs a sharing (authenticated, in the malicious setting) of that bit.

Functionality $\mathcal{F}^b_{\mathbf{const}} \to \langle b \rangle$

Output: The functionality does the following:

1. Choose bit $r \xleftarrow{\$} \{0,1\}$.
2. (Semi-honest setting) Output r to P_i and $r \oplus b$ to P_j.
3. (Malicious setting) Construct authenticated bits $r_i = \langle r \rangle^{(i)}$ and $r_j = \langle r \oplus b \rangle^{(j)}$, and output r_i to party P_i and r_j to party P_j.

In the semi-honest setting, this functionality can be instantiated by having P_i generate a random bit r and sending r to P_j, who computes $r \oplus b$. In the malicious setting we can instantiate this using the protocol described in [NNOB12, Figure 3].

Bit secret sharing. The functionality $\mathcal{F}_{\mathbf{ss}}$ in the semi-honest setting is the standard secret sharing functionality. In the malicious setting, the functionality creates an authenticated sharing of the input bit.

Functionality $\mathcal{F}^i_{\mathbf{ss}}(b) \to \langle b \rangle$

Input: Party P_i inputs a bit b.

Output: The functionality does the following:

1. Select $r \xleftarrow{\$} \{0,1\}$ uniformly at random.
2. (Semi-honest setting) Output r to party P_j, and $b \oplus r$ to party P_i.
3. (Malicious setting) Construct authenticated bits $b_j = \langle r \rangle^{(j)}$ and $b_i = \langle b \oplus r \rangle^{(i)}$, and output b_j to party P_j and b_i to party P_i.

Implementing the functionality in the semi-honest setting is trivial. In the malicious setting we can use the **Input** protocol described in [NNOB12, Figure 6].

One-out-of-two oblivious secret sharing. The functionality $\mathcal{F}_{\mathbf{oshare}}$ is used to share the sub-keys of the garbled table in an oblivious fashion while preserving consistency with respect to the circuit such that the circuit evaluation succeeds given the correct input sub-keys. More precisely, $\mathcal{F}_{\mathbf{oshare}}$ interacts with two parties, called the *sender* P_j and the *receiver* P_i; it expects two inputs, m_0 and m_1, from the sender along with a two-out-of-two sharing $\langle b \rangle$ of a selection bit b between the sender and receiver, and outputs a random two-out-of-two sharing $[m_b]$ of m_b. In the malicious setting, $\langle b \rangle$ is an authenticated bit share. The functionality does not leak any information about b to the parties. Furthermore, when the sender is honest, it leaks no information on its inputs to the receiver. However, to ensure simulatability we allow a corrupted sender to choose his output share, $y^{(j)}$, at will.

Functionality $\mathcal{F}^{i,j}_{\mathbf{oshare}}(\langle b\rangle^{(i)}, (\langle b\rangle^{(j)}, m_0, m_1, y^{(j)})) \to [m_b]$

Input: Party P_i inputs share $b_i = \langle b\rangle^{(i)}$. Party P_j inputs share $b_j = \langle b\rangle^{(j)}$ and vector $(m_0, m_1) \in \{0,1\}^k \times \{0,1\}^k$. In the malicious setting, b_i and b_j are authenticated. Additionally, party P_j input a value $y^{(j)} \in \{0,1\}^k \cup \{\perp\}$; if P_j is honest he sets $y^{(j)} = \perp$, otherwise $y^{(j)}$ can be arbitrary.

Output: The functionality does the following:

1. (Malicious setting) If either $\langle b\rangle^{(i)}$ or $\langle b\rangle^{(j)}$ is not a correctly authenticated bit then abort.
2. Compute $b = b_i \oplus b_j$
3. If $y^{(j)} = \perp$, then choose $y^{(j)} \xleftarrow{\$} \{0,1\}^k$.
4. Output $m_b \oplus y^{(j)}$ to P_i and $y^{(j)}$ to P_j.

See Appendix B for an instantiation of $\mathcal{F}_{\mathbf{oshare}}$ in the semi-honest setting, and Appendix C for an instantiation in the malicious setting.

Oblivious secret sharing. The oblivious secret sharing functionality $\mathcal{F}_{\mathbf{rand}}$ takes no inputs, and outputs a sharing of a random bit. In the malicious setting, this output sharing is authenticated.

Functionality $\mathcal{F}_{\mathbf{rand}} \to \langle r\rangle$

Output: The functionality does the following:

1. Choose bits $r, r' \xleftarrow{\$} \{0,1\}$.
2. (Semi-honest setting) Output r' to P_i and $r \oplus r'$ to P_j.
3. (Malicious setting) Construct authenticated bits $r_i = \langle r'\rangle^{(i)}$ and $r_j = \langle r \oplus r'\rangle^{(j)}$, and output r_i to party P_i and r_j to party P_j.

In the semi-honest setting, this can be easily instantiated by each party choosing a random bit r_i and letting $r = r_i \oplus r_j$. In the malicious setting we need to construct authenticated shares and thus need additional machinery: we can utilize the **Rand** protocol in [NNOB12, Figure 6].

Oblivious gate evaluation. The functionality $\mathcal{F}^{G}_{\mathbf{gate}}$ takes as inputs shares of bits a and b, and outputs a share of $G(a,b)$ for some binary gate G. In the malicious setting, both input and output shares are authenticated.

Functionality $\mathcal{F}^{G}_{\mathbf{gate}}(\langle a\rangle, \langle b\rangle) \to \langle G(a,b)\rangle$

Input: The parties input bit shares $\langle a\rangle$ and $\langle b\rangle$. In the malicious setting, these shares are authenticated.

Auxiliary Input: The description of a binary gate G.

Output: The functionality does the following:

1. (Malicious setting) If any of the shares are not correctly authenticated then abort.
2. Compute a and b from the shares.
3. Compute $c = G(a,b)$ and output a sharing (authenticated in the malicious setting) of c.

As our circuits require only AND and XOR gates, we only consider those choices for G here. In the semi-honest setting, we can compute XOR gates locally and AND gates using one-out-of-four OT, as is done in the GMW protocol [GMW87]. In the malicious setting we can efficiently instantiate the functionality for XOR and AND gates using the **XOR** and **AND** protocols in [NNOB12, Figure 6].

Oblivious transfer. The functionality $\mathcal{F}_{\mathbf{ot}}$ implements standard oblivious transfer.

Functionality $\mathcal{F}_{\mathbf{ot}}^{i,j}(b, (m_0, m_1)) \to m_b$

Input: Party P_i inputs a choice bit b. Party P_j inputs two messages m_0 and m_1.

Output: The functionality outputs m_b to P_i, and $\perp$ to P_j.

This can be implemented efficiently in both the semi-honest and malicious setting using known existing protocols (e.g., [PVW08]).

B Hybrid Implementations for the Semi-honest Protocol

One-out-of-two oblivious secret sharing. The following protocol implements the $\mathcal{F}_{\mathbf{oshare}}$ functionality:

Protocol $\Pi_{\mathbf{oshare}}^{i,j}(\langle b\rangle^{(i)}, (\langle b\rangle^{(j)}, m_0, m_1, \perp))$

Input: Party P_i inputs his share $b_i = \langle b\rangle^{(i)}$ of $\langle b\rangle$. Party P_j inputs his share $b_j = \langle b\rangle^{(j)}$ of $\langle b\rangle$ along with two strings $m_0, m_1 \in \{0,1\}^k$.

1. P_j chooses $r \xleftarrow{\$} \{0,1\}^k$ uniformly at random.
2. Execute $\mathcal{F}_{\mathbf{ot}}$ with P_j as the sender having inputs $(s_0, s_1) = (m_0 \oplus r, m_1 \oplus r)$, and P_i as the receiver having input $b' = 1 \oplus b_i$; denote P_i's output as y_i.

Outputs: P_j outputs $y_j = ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r$ and P_i outputs y_i.

Lemma 1. *The protocol $\Pi_{\mathbf{oshare}}^{i,j}$ securely implements the functionality $\mathcal{F}_{\mathbf{oshare}}^{i,j}$ in the presence of a semi-honest adversary in the $\mathcal{F}_{\mathbf{ot}}$-hybrid world.*

Proof. First, we show correctness; namely, we argue that the output of the protocol is a two-out-of-two sharing of m_b, that is, $y_i \oplus y_j = m_b$. Indeed,

$$\begin{aligned} y_i \oplus y_j &= m_{1\oplus b_i} \oplus r \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r \\ &= m_{1\oplus b_i} \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)). \end{aligned}$$

Note that if $b_j = 0$, we have

$$y_i \oplus y_j = m_{1\oplus b_i} \oplus m_0 \oplus m_1 = m_b.$$

Likewise, if $b_j = 1$, we have

$$y_i \oplus y_j = m_{1\oplus b_i} = m_b.$$

To prove that the protocol is simulatable, observe that (1) P_j receives no information in the protocol (which follows from the privacy of OT) and (2) P_i only sees y_i, where the value $m_{1\oplus b_i}$ is perfectly blinded by r. Hence, similarly to the ideal evaluation of $\mathcal{F}_{\mathbf{oshare}}^{i,j}$, the values seen by the parties give them no information. More formally, we consider the following cases:

P_i is corrupted: The simulator, emulating the execution of $\mathcal{F}_{\mathbf{ot}}$ towards P_i, waits for $\mathcal{A}$ to input his bit $1 \oplus \langle b\rangle^{(i)}$. The simulator extracts $\langle b\rangle^{(i)}$, submits it to $\mathcal{F}_{\mathbf{oshare}}$, forwards the reply to $\mathcal{A}$, and halts with $\mathcal{A}$'s output.

P_j **is corrupted:** The simulator receives the messages $(m_0 \oplus r, m_1 \oplus r)$ from $\mathcal{A}$ and extracts r, which is possible due to the semi-honest setting. The simulator then submits $(\langle b \rangle^{(j)}, m_0, m_1, y^{(j)})$ to $\mathcal{F}_{\mathbf{oshare}}$, where $y^{(j)} = (m_0 \oplus m_1) \odot (1 \oplus \langle b \rangle^{(j)}) \oplus r$, and halts with $\mathcal{A}$'s output.

Noting that each of these simulators perfectly simulates the adversary in the $\mathcal{F}_{\mathbf{ot}}$-hybrid world, the protocol is secure. □

C Hybrid Implementations for the Malicious Protocol

In the malicious setting we utilize ideas from the NNOB protocol [NNOB12]. In particular, each party P_i holds a global key Δ_i, which they use to construct authenticated bit shares. For a bit b authenticated towards P_i, P_i holds both the bit b and a MAC M_b, with P_j holding the authentication key K_b, with the condition that $M_b = K_b \oplus b\Delta_j$. To ease notation, in this section we let $\langle b \rangle^{(i)} = (b, M_b, K_b)$.

Authenticated bit. We repeat here the functionality $\mathcal{F}_{\mathbf{aBit}}$ from [NNOB12, Figure 5].

Functionality $\mathcal{F}_{\mathbf{aBit}} \to \langle r \rangle^{(i)}$

Auxiliary Input: Party P_j inputs his global key $\Delta_j \leftarrow \{0,1\}^k$.

Output:
1. (If P_i is malicious) Set $\langle b \rangle^{(i)} = (b, M, M \oplus b\Delta)$.
2. (If P_j is malicious) Let $b \xleftarrow{\$} \{0,1\}$ and set $\langle b \rangle^{(i)} = (b, K \oplus b\Delta_j, K)$.
3. (If both are honest) Let $b \xleftarrow{\$} \{0,1\}$ and $K \xleftarrow{\$} \{0,1\}^k$, and set $\langle b \rangle^{(i)} = (b, K \oplus b\Delta_j, K)$.
4. Output $(b, K \oplus b\Delta_j)$ to P_i and (K, Δ_j) to P_j.

The implementation of $\mathcal{F}_{\mathbf{aBit}}$ is detailed in [NNOB12, Section 4] and not repeated here. Note that the parties can utilize $\mathcal{F}_{\mathbf{aBit}}$ to construct a constant bit by P_i setting $M = 0^k$ and P_j setting $K = b\Delta_j$.

Receiver-authenticated one-out-of-two oblivious transfer. We first define the functionality for receiver-authenticated oblivious transfer:

Functionality $\mathcal{F}^{i,j}_{\mathbf{raot}}(\langle b \rangle^{(i)}, (m_0, m_1)) \to m_b$

Input: Party P_i inputs an authenticated choice bit b. Party P_j inputs two messages m_0 and m_1.

Output: The functionality does the following:
1. If P_i's choice bit b is not correctly authenticated then abort.
2. Output m_b to P_i.

In order to efficiently implement $\mathcal{F}_{\mathbf{raot}}$, we need the following functionality:

Functionality $\mathcal{F}_{\mathbf{eq}}(a, b) \to \{0,1\}$

Input: Party P_i inputs $a \in \{0,1\}^k$, and party P_j inputs $b \in \{0,1\}^k$.

Output: The functionality outputs 1 if $a = b$, and 0 otherwise.

This can be instantiated efficiently using two calls to a random oracle H; see [NNOB12, pg. 7]. We can thus instantiate $\mathcal{F}_{\mathbf{raot}}$ as follows:

Protocol $\Pi_{\mathbf{raot}}(\langle b\rangle^{(i)}, (m_0, m_1))$

Let $\langle b\rangle^{(i)} = (b, M_b, K_b)$.

1. The parties compute $\langle r\rangle^{(i)} = (r, M_r, K_r) \leftarrow \mathcal{F}^i_{\mathbf{aBit}}$.
2. P_i computes $d = b \oplus r$ and sends d to P_j.
3. P_i sends $M_b \oplus M_r$ to $\mathcal{F}_{\mathbf{eq}}$, and P_j sends $(K_b \oplus K_r) \oplus d\Delta_j$ to $\mathcal{F}_{\mathbf{eq}}$, to check the equality of the two values. If they are not equal, P_j aborts the protocol.
4. The parties then compute $\langle d\rangle^{(i)} \leftarrow \mathcal{F}^d_{\mathbf{const}}$.
5. Let $\langle w\rangle^{(i)} = \langle r\rangle^{(i)} \oplus \langle d\rangle^{(i)}$. P_j sends $X_0 = H(K_w) \oplus m_0$ and $X_1 = H(K_w \oplus \Delta_j) \oplus m_1$ to P_i.

Output: P_i outputs $X_w \oplus H(M_w)$ and P_j outputs $\perp$.

Lemma. *The protocol* $\Pi_{\mathbf{raot}}$ *securely implements the functionality* $\mathcal{F}_{\mathbf{raot}}$ *in the presence of a malicious adversary in the Random Oracle model.*

Proof. Noting that $w = b$ in the protocol, correctness is immediate. To prove simulatability, we consider the following corruption cases:

P_i **is corrupted:** The simulator $\mathcal{S}$ simulating an adversary $\mathcal{A}$ corrupting P_i proceeds as follows. $\mathcal{S}$ forwards its input (b, M_b) to $\mathcal{A}$ as input to the protocol. If $\mathcal{A}$ sends a message v to $\mathcal{F}_{\mathbf{eq}}$ such that $v \neq M_b \oplus M_r$, $\mathcal{S}$ aborts the protocol. Otherwise, $\mathcal{S}$ sends $\langle b\rangle^{(i)}$ to the trusted third party, receiving back m_b. $\mathcal{S}$ then generates two random bit-strings X_0 and X_1, and programs H so that $H(M_w) = X_b \oplus m_b$. Finally, $\mathcal{S}$ sends X_0 and X_1 to $\mathcal{A}$.

P_j **is corrupted:** The simulator $\mathcal{S}$ simulating an adversary $\mathcal{A}$ corrupting P_j proceeds as follows. $\mathcal{S}$ forwards its input (m_0, m_1, K_b) to $\mathcal{A}$ as input to the protocol. If $\mathcal{A}$ sends a message v to $\mathcal{F}_{\mathbf{eq}}$ such that $v \neq (K_b \oplus K_r) \oplus d\Delta_j$, $\mathcal{S}$ aborts the protocol. Next, $\mathcal{S}$ waits until P_j sends X_0 and X_1 to P_i. It then uses its knowledge of $\langle r\rangle^{(i)}$ and $\langle d\rangle^{(i)}$ to extract m_0 and m_1, which it feeds to the trusted third party.

Noting that each of these simulators perfectly simulate the adversary in the $(\mathcal{F}_{\mathbf{aBit}}, \mathcal{F}_{\mathbf{const}}, \mathcal{F}_{\mathbf{eq}})$-hybrid world, the protocol is secure. □

One-out-of-two oblivious secret sharing. We can instantiate $\mathcal{F}_{\mathbf{oshare}}$ in the malicious setting using a protocol similar to $\Pi_{\mathbf{oshare}}$ with the following modifications:

1. We use receiver-authenticated OT in place of standard OT;
2. To ensure that the simulator can extract consistent inputs from a corrupted sender, we do two invocations of $\mathcal{F}_{\mathbf{raot}}$;
3. In order to extract P_j's input in the case P_j is corrupt, we need an addition authentication check requiring one invocation of $\mathcal{F}_{\mathbf{aBit}}$.

Protocol $\Pi^{i,j}_{\mathbf{oshare-m}}(\langle b\rangle^{(i)}, (\langle b\rangle^{(j)}, m_0, m_1, \perp))$

Let $\langle b \rangle^{(i)} = (b_i, M_{b_i}, K_{b_i})$ and $\langle b \rangle^{(j)} = (b_j, M_{b_j}, K_{b_j})$.

1. P_j chooses $r_0, r_1 \xleftarrow{\$} \{0,1\}^k$ uniformly at random.
2. Execute $\mathcal{F}_{\mathbf{raot}}$ with P_j as sender having inputs $(s_0, s_1) = (m_0 \oplus r_0, m_1 \oplus r_1)$, and P_i as the receiver having input $\langle 1 \rangle^{(i)} \oplus \langle b \rangle^{(i)}$; P_i denotes his output by y_i.
3. Execute $\mathcal{F}_{\mathbf{raot}}$ with P_j as sender having inputs $(s_0, s_1) = (r_0, r_1)$, and P_i as the receiver having input $\langle b \rangle^{(i)}$; P_i denotes his output by r_{b_i}.
4. Execute $\langle r \rangle^{(j)} = (r, M_r, K_r) \leftarrow \mathcal{F}^j_{\mathbf{aBit}}$. P_j sends (d, M_d) to P_i, where $d = r \oplus b_j$ and $M_d = M_r \oplus M_{b_j}$, and P_i checks if $(d, M_d, K_d \oplus K_r)$ is a valid authenticated bit, aborting if not.

Outputs: P_j outputs $y^{(j)} = ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r_0 \oplus r_1$ and P_i outputs $y_i \oplus r_{b_i}$.

Lemma. *The protocol $\Pi_{\mathbf{oshare-m}}$ securely implements the functionality $\mathcal{F}_{\mathbf{oshare}}$ in the presence of a malicious adversary in the ($\mathcal{F}_{\mathbf{aBit}}$, $\mathcal{F}_{\mathbf{raot}}$)-hybrid world.*

Proof. We first demonstrate correctness. It suffices to show that $y_i \oplus y_j = m_b$. Indeed,

$$\begin{aligned} y_i \oplus y_j &= (m_{1 \oplus b_i} \oplus r_{1 \oplus b_i} \oplus r_{b_i}) \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)) \oplus r_0 \oplus r_1 \\ &= m_{1 \oplus b_i} \oplus ((m_0 \oplus m_1) \odot (1 \oplus b_j)), \end{aligned}$$

and the derivation follows exactly as in the semi-honest case.

To prove that the protocol is simulatable, we consider the following corruption cases:

P_i **is corrupted:** The simulator $\mathcal{S}$ waits for $\mathcal{A}$ to input his choice bit $\langle b' \rangle^{(i)}$ to the first $\mathcal{F}_{\mathbf{raot}}$ invocation. If $\langle b' \rangle$ is not a valid authenticated bit, $\mathcal{S}$ aborts. Otherwise, $\mathcal{S}$ returns to $\mathcal{A}$ a random string y'_i. Likewise, in the second $\mathcal{F}_{\mathbf{raot}}$ invocation, $\mathcal{S}$ retrieves $\langle b \rangle^{(i)}$ from $\mathcal{A}$, aborting if the authentication check fails. $\mathcal{S}$ then invokes $\mathcal{F}_{\mathbf{oshare}}$ with $\langle b \rangle^{(i)}$, receiving P_i's output y_i. $\mathcal{S}$ computes $r'_i = y_i \oplus y'_i$ and sends r'_i to $\mathcal{A}$ as the output of the second $\mathcal{F}_{\mathbf{raot}}$. Finally, $\mathcal{S}$ acts as P_j would in Step 4, and halts with $\mathcal{A}$'s output.

P_j **is corrupted:** The simulator $\mathcal{S}$ emulates the two executions of $\mathcal{F}_{\mathbf{raot}}$ towards the adversary $\mathcal{A}$ controlling P_j, from which $\mathcal{S}$ receives (x_0, x_1) and (r'_0, r'_1), respectively. In Step 4, $\mathcal{S}$ extracts $\langle b_j \rangle^{(j)}$. $\mathcal{S}$ then computes $(m'_0, m'_1) = (x_0 \oplus r'_0, x_1 \oplus r'_1)$ and submits the message $(\langle b_j \rangle^{(j)}, m'_0, m'_1, y^{(j)})$ to $\mathcal{F}_{\mathbf{oshare}}$, where $y^{(j)} = ((m'_0 \oplus m'_1) \odot (1 \oplus b_j)) \oplus r'_0 \oplus r'_1$, and halts with $\mathcal{A}$'s output.

Noting that each of these simulators perfectly simulates the adversary in the ($\mathcal{F}_{\mathbf{aBit}}$, $\mathcal{F}_{\mathbf{raot}}$)-hybrid world, the protocol is secure. □

D 2PC Protocol Using Distributed Garbled Circuits

D.1 The Semi-Honest Setting

We describe a secure two-party computation protocol $\Pi_{\mathbf{2PC}}(P_1, P_2)$ which uses our two-party distributed garbling scheme $\Pi_{\mathbf{GC}}(P_1, P_2)$ as a sub-protocol to securely compute any polynomial-size circuit C toward P_2; see Figure 3 for the detailed description. The parties first perform a distributed garbling of the circuit and secret-sharing of their inputs. The parties then exchange input keys as follows:

Protocol $\Pi_{\mathbf{2PC}}(P_1, P_2)$

Auxiliary Inputs: Security parameter k, circuit $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G) \leftarrow C$.

Inputs: For $w \in \{1, \ldots, \mathsf{n}_1\}$, P_1 has inputs b_w; for $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$, P_2 has inputs b_w.

1. The parties execute $\Pi_{\mathbf{GC}}(P_1, P_2)$.
2. For $w \in \{1, \ldots, \mathsf{n}_1\}$: The parties execute $\langle b_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(b_w)$.
3. For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: The parties execute $\langle b_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(b_w)$.
4. P_1 sends GC^1 to P_2.
5. For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sends $(s^1_{w,b_w \oplus \lambda_w}, \langle b_w \rangle^{(1)} \oplus \langle \lambda_w \rangle^{(1)})$ to P_2 who reconstructs $b_w \oplus \lambda_w$ locally.
6. For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$: P_2 sends $\langle b_w \rangle^{(2)} \oplus \langle \lambda_w \rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_w$ locally. P_1 then sends $s^1_{w,b_w \oplus \lambda_w}$ to P_2.
7. P_2 evaluates the garbled circuit using the keys $(s^1_{w,b_w \oplus \lambda_w}, s^2_{w,b_w \oplus \lambda_w})$ and selector bits $b_w \oplus \lambda_w$, for $w \in \{1, \ldots, \mathsf{n}\}$.

Figure 3: Two-party secure function evaluation.

- For each of P_1's input wires, P_1 sends to P_2 the appropriate sub-key $s^1_{w,b_w \oplus \lambda_w}$, as well as his share $\langle b_w \rangle^{(1)} \oplus \langle \lambda_w \rangle^{(1)}$; P_2 reconstructs $b_w \oplus \lambda_w$ and selects sub-key $s^2_{w,b_w \oplus \lambda_w}$ to use for decryption.[4]
- For each of P_2's input wires, we do a similar protocol as for P_1's input wires. Here, P_2 sends $\langle b_w \rangle^{(2)} \oplus \langle \lambda_w \rangle^{(2)}$ to P_1, who reconstructs $b_w \oplus \lambda_w$; P_1 then sends sub-key $s^1_{w,b_w \oplus \lambda_w}$ to P_2.[5]

Given the "shares" of the garbled circuit and the input keys, P_2 can evaluate the garbled circuit to retrieve the output.

Theorem 3. *The protocol $\Pi_{\mathbf{2PC}}(P_1, P_2)$ securely evaluates the circuit C in the presence of a static, semi-honest adversary in the $(\mathcal{F}_{\mathbf{const}}, \mathcal{F}_{\mathbf{gate}}, \mathcal{F}_{\mathbf{oshare}}, \mathcal{F}_{\mathbf{ot}}, \mathcal{F}_{\mathbf{rand}}, \mathcal{F}_{\mathbf{ss}})$-hybrid world.*

Proof. Correctness of the construction can be easily verified by inspection of the protocol. In the remainder of the proof we describe (black-box straight-line) simulators $\mathcal{S}_1$ and $\mathcal{S}_2$ which simulate an adversary corrupting P_1 and P_2, respectively.

The simulator $\mathcal{S}_1$: We simulate adversary P_1^* corrupting P_1 as follows. $\mathcal{S}_1$ plays as P_2 and follows the protocol description using $0^{\mathsf{n}-\mathsf{n}_1}$ as input. The messages which P_1^* sees during the simulation are all shares of random values and input values. Therefore, due to the property of the secret sharing, the simulation is perfect.

The simulator $\mathcal{S}_2$: Simulating adversary P_2^* corrupting P_2 is less trivial. In addition to running $\Pi_{\mathbf{GC}}$, $\mathcal{S}_2$ must also simulate sending P_1's share of the garbled circuit along with the correct keys and mask bits that P_1 sends to P_2^* after the execution of $\Pi_{\mathbf{GC}}$. However, $\mathcal{S}_2$ does not know the inputs of P_1. Hence, $\mathcal{S}_2$ has to come up with a sharing of a "fake" circuit which is indistinguishable from the original circuit. We handle this as follows. $\mathcal{S}_2$ uses 0 for each of P_1's input bits, which would cause the output to be $f(0^{\mathsf{n}_1}, y)$. While this is not necessarily the correct output (as P_1's input could be any $x \leftarrow \{0,1\}^{\mathsf{n}_1}$), $\mathcal{S}_2$ can correct this *by modifying the selection bits on the output gates of the circuit* to contain $z = f(x, y)$ rather than $f(0^{\mathsf{n}_1}, y)$. The security of the encryption scheme

[4]Note that in the semi-honest setting P_1 can in fact just send $b_w \oplus \lambda_w$ instead of sending his share; however, when we discuss malicious security we will find it necessary for P_1 to send his share and *not* $b_w \oplus \lambda_w$. Thus, we include this slight complication in the semi-honest setting to help ease our transition to malicious security.

[5]Note that the prior footnote about sending $b_w \oplus \lambda_w$ instead of the shares applies in this step as well.

guarantees that these modified selection bits do not provide the adversary with any distinguishing advantage.

More precisely, playing as P_1 and emulating the ideal functionalities, $\mathcal{S}_2$ proceeds as follows:

1. As $\mathcal{S}_2$ does not know P_1's input, it uses 0^{n_1} instead.

2. S_2 sends P_2^*'s input y to the SFE ideal functionality and receives the function output $z = f(x, y)$ in return.

3. In performing the distributed garbling, for each of P_2^*'s input wires, $\mathcal{S}_2$ receives the mask bits λ_w (through the calls to $\mathcal{F}_{\mathbf{ss}}$), and for each gate G_γ, $\mathcal{S}_2$ receives the sub-keys $s^2_{\gamma,0}$ and $s^2_{\gamma,1}$ (through the calls to $\mathcal{F}_{\mathbf{oshare}}$). $\mathcal{S}_2$ chooses all other mask bits uniformly at random except for the output masks (which $\mathcal{S}_2$ sets to 0).

   Now, when the output wires are decrypted, P_2^* expects to receive the correct output z. $\mathcal{S}_2$ accomplishes this as follows. Given inputs x and y, $\mathcal{S}_2$ can compute, for each gate G_γ, the expected output z_γ of that gate. Thus, for every gate G_γ in the circuit, $\mathcal{S}_2$ modifies the construction of the garbled gate as follows: For each row $P[\gamma, j, k]$ of the garbled gate, $\mathcal{S}_2$ modifies the output of the $\mathcal{F}_{\mathbf{gate}}$ calls to output $\langle z_\gamma \rangle = (\langle z_\gamma \rangle^{(1)}, \langle z_\gamma \rangle^{(2)})$ (instead of having $\mathcal{F}_{\mathbf{gate}}$ output $\langle \sigma_{\gamma,j,k} \rangle$). This ensures that irrespective of which row is actually decrypted, the output of gate G_γ will be z_γ as in the ideal computation, and the sub-keys will be consistent with the selector bit.

This concludes the description of the simulator. We now argue that the simulated transcript is indistinguishable from the real one. The proof of simulation follows by a hybrid argument. As in [LP09] we assume that all the gates have two inputs and a single output, and are topologically ordered, where the input gates come first, then the inner gates and finally the output gates.

For $i \in \{0, \ldots, \mathsf{q}\}$ we define the hybrid H_i as follows: H_i is the protocol execution in which the garbled gates indexed by $\mathsf{n}+1, \ldots, \mathsf{n}+i$ are constructed according to the real execution and the remaining $\mathsf{q}-i$ gates constructed according to the simulated execution. Note that H_0 corresponds to the real execution, and H_{q} to the simulated execution. So, it suffices to show that for $i \in \{0, \ldots, \mathsf{q}-1\}$, the neighboring hybrids H_i and H_{i+1} are indistinguishable.

Choose i which maximizes the adversary's advantage, and let G_γ refer to $G_{\mathsf{n}+i}$ (namely, G_γ is the gate which is garbled as in the real execution in hybrid H_i but replaced by a simulated gate in hybrid H_{i+1}). Denote by α and β the input wires to G_γ and by γ the output wire. In H_i, the third component of the plaintext in the garbled row $P^1[\gamma, j, k]$ is $\langle \sigma_{\gamma,j,k} \rangle^{(1)}$, whereas in H_{i+1} this component is $\langle z_\gamma \rangle^{(1)}$. The claim that this does not create a distinguishing advantage follows from the security of the encryption scheme. The argument is nearly identical to the argument in [LP09, pp. 183–187], and is thus not repeated.

This completes the proof. □

D.2 The Malicious Setting

Intuitively, $\Pi_{\mathbf{2PC}}(P_1, P_2)$, using the "authenticated" ideal functionalities from Section 3.3, is a maliciously secure protocol because the adversary is unable to cheat by modifying the authenticated shared bits. The only alternative is for the adversary to cheat by modifying his sub-keys, which would cause the protocol to abort. However, each garbled gate is permuted according to random shared bits, and therefore the adversary has no notion of which row will be decrypted on evaluation;

if he corrupts t rows in a given garbled gate, the probability of the protocol aborting is $t/4$, *independent of each party's input.*

Theorem 4. *Let C be an arbitrary polynomial-size circuit. Then the protocol $\Pi_{\mathbf{2PC}}(P_1, P_2)$, using authenticated hybrids as described above, securely evaluates the circuit C in the presence of a (static) malicious adversary in the $(\mathcal{F}_{\mathbf{const}}, \mathcal{F}_{\mathbf{gate}}, \mathcal{F}_{\mathbf{oshare}}, \mathcal{F}_{\mathbf{rand}}, \mathcal{F}_{\mathbf{ss}})$-hybrid world.*

Proof. We construct simulators $\mathcal{S}_1$ and $\mathcal{S}_2$ which simulate an adversary corrupting P_1 and P_2, respectively. In the following analysis, we ignore the negligible probability difference due to the direct attack on the authentication mechanism.

Simulator $\mathcal{S}_1$: We simulate adversary P_1^* corrupting party P_1 as follows. The simulator $\mathcal{S}_1$ chooses 0^{n-n_1} as P_2's input and then runs exactly as P_2 would. In addition, $\mathcal{S}_1$ extracts P_1^*'s input x through the calls to $\mathcal{F}_{\mathbf{ss}}^1$ and passes x to the ideal SFE functionality.

As $\mathcal{S}_1$ acts exactly as P_2 does (albeit on a different input), and P_1^* receives no output, we need only show that the protocol aborts with equal probability across the two views.

Note that P_1^* has three possible places in the protocol in which he can try to force the protocol to abort:

1. Sending an invalid sub-key in Step 5 or Step 6 of $\Pi_{\mathbf{2PC}}$,
2. Inputting invalid or flipped sub-keys into the calls to $\mathcal{F}_{\mathbf{oshare}}$, or
3. Encrypting the incorrect sub-key shares or using some arbitrary string as encryptions.

We claim that the probability of aborting due to any of the above attacks is *independent* of P_2's input. Clearly, if P_1^* sends invalid sub-keys for his own input wires, the probability of aborting is independent of P_2's input. In the case that P_1^* sends an invalid sub-key in Step 6 of $\Pi_{\mathbf{2PC}}$, the probability of aborting is independent of P_2's input due to the masking by the (uniformly chosen) mask bit.

Now consider the case where P_1^* corrupts t rows in a given garbled gate. Note that even though P_1^* can control *which* rows in the garbled gate table to corrupt, the probability that any given row is hit during evaluation is exactly $1/4$ (by the security of the point-and-permute method). Thus, the probability that a given bad row is hit is $t/4$, independent of the bits on the incoming wires into the gate. Thus, as the probability of aborting is independent of P_2's input, the two views are perfectly indistinguishable.

Simulator $\mathcal{S}_2$: We simulate adversary P_2^* corrupting party P_2 as follows. The simulator $\mathcal{S}_2$ selects 0^{n_1} as the input of P_1, and then proceeds to act as P_1 in $\Pi_{\mathbf{GC}}$. Then, $\mathcal{S}_2$ extracts P_2^*'s input y through the calls to $\mathcal{F}_{\mathbf{ss}}^2$ and passes y to the ideal functionality, receiving back $f(x, y)$. Now, $\mathcal{S}_2$ continues executing as P_1, except it modifies its share for the output of $\mathcal{F}_{\mathbf{gate}}$ on the output gates so that the selector bits for all rows in the output gates contain the appropriate bit from $f(x, y)$. In more detail, right before sending its share GC^1 at Step 4 of the protocol, $\mathcal{S}_2$ does the following:

- For each output gate G_γ, let z_γ denote the correct output (i.e., the appropriate bit from $f(x, y)$) of the gate. Now, for each of the four rows of this gate, $\mathcal{S}_2$ modifies the original authenticated sharing $(\langle z_{\gamma,1}\rangle^{(1)}, \langle z_{\gamma,2}\rangle^{(2)})$ into a new sharing $(\langle z^*_{\gamma,1}\rangle^{(1)}, \langle z_{\gamma,2}\rangle^{(2)})$ that would be reconstructed to z_γ. Note that this is possible, since $\mathcal{S}_2$ has been emulating the $\mathcal{F}_{\mathbf{gate}}$ functionality and has all the information necessary to construct new authenticated shares. In addition, $\mathcal{S}_2$ modifies the corresponding encryption in the garbled gate accordingly.

$\mathcal{S}_2$ continues executing as P_1, and outputs whatever P_2^* outputs.

We now prove that the view of the adversary when communicating with $\mathcal{S}_2$ versus the view when communicating with a real P_1 is computationally indistinguishable. We show this by constructing a set of hybrids and proving indistinguishability between them.

- H_0: The same as the real execution with P_1.
- H_1: The same as H_0, except the output of $\mathcal{F}_{\mathbf{gate}}$ in each output gate is modified to be equal to an authenticated sharing of the correct output from $f(x, y)$.
- H_2: The same as H_1, except input 0^{n_1} is used instead of P_1's real input.

We first show that hybrids H_0 and H_1 are computationally indistinguishable. This follows from the security of the underlying garbling scheme; the only difference here is that P_2^* can try to force the protocol to abort. However, by the security of the authenticated bit sharing scheme, the output of $\mathcal{F}_{\mathbf{gate}}$ towards P_2^* provides no information about the underlying selector bit's value, and thus P_2^* acts independently of the value $\sigma_{\gamma,i,j}$.

The indistinguishability between H_1 and H_2 can be shown similarly to the case in which P_1 is corrupted.

Note that H_0 is simply the real execution of the protocol, and H_2 is the execution of the simulator $\mathcal{S}_2$. Thus, as each hybrid is (at least) computationally indistinguishable from its neighboring hybrids, we conclude that the protocol is secure. □

E Proof of Theorem 1

Theorem. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof (Sketch). The proof is similar to prior work in two-party garbling schemes based on cut-and-choose, such as [LP07], [LP11], and [sS11].

We make use of the following lemma:

Lemma. *Consider garbled gate G_γ with input wires α and β, and let $X_{w,b} = (s^1_{w,b}, s^2_{w,b}, b \oplus \lambda_w)$, for $w \in \{\alpha, \beta\}$ denote the valid keys. Fix a (valid) key $X_{w,b}$ for some fixed w and b. Let $\bar{X}_{w,b}$ be equal to $X_{w,b}$ except that two of the three components (i.e., the sub-keys and selector bit) are altered arbitrarily. Then using key $\bar{X}_{w,b}$ to decrypt G_γ causes a decryption failure with all but negligible probability.*

Proof. This follows directly from our encryption scheme and garbling scheme. □

What this lemma says is that for a given garbled gate, a sub-key / selector bit combo can only be used correctly on a single row of the garbled table, and modifying some (but not all) of the components results in a decryption failure; thus an adversary must change *both* the sub-key and permutation bit accordingly for the garbled gate to successfully decrypt. Note that in the two-party secure computation protocol described in Section 3 we enforce the above by authenticating all of the selector bits (thus preventing any malicious party from altering these). However, the authenticated sharing protocol only works between two parties, namely the parties doing the distributed garbling. Thus, we need a way for the evaluator to gain confidence in the sub-key / selector bit combos sent

to him by P_1 and P_2. We do this by utilizing the Diffie-Hellman pseudorandom synthesizer trick of [LP11]. This enforces that P_1 and P_2 are consistent in the sub-keys they send to P_3, and because at least one sub-key is correct, the adversary can at most cause P_3 to abort.

There are six possible (interesting) corruption cases. However, due to symmetries, we only need to consider four of them.

The adversary corrupts parties P_1 and P_2. We need to construct a simulator $\mathcal{S}$ with access to the adversary $\mathcal{A}$ (who controls P_1 and P_2) and a trusted third party which computes $f(x, y, z)$ given inputs x, y, and z. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and the runs as P_3 would until Step 10. Here, $\mathcal{S}$ uses the witnesses a^i_{w,b_w} send by P_1 and P_2 to $\mathcal{F}_{\mathbf{zkpok}}$ to extract their inputs. $\mathcal{S}$ then feeds these inputs to the trusted third party, receiving back $f(x, y, z)$. $\mathcal{S}$ continues to run as P_3 would, and halts, outputting whatever $\mathcal{A}$ outputs.

We now argue that the adversary's view in the real and ideal worlds are computationally indistinguishable. The proof closely follows that shown in [LP11, pp. 17–21], and thus we only give some intuition here.

Note that if $\mathcal{A}$ tries to cheat in Step 5, he will get caught in the cut-and-choose step with high probability. Similarly, if $\mathcal{A}$ tries to send different keys in Step 8 (i.e., the "input inconsistency" attack), he will get caught in Step 10 when proving the consistency of the sub-keys sent.

The adversary corrupts parties P_1 and P_3. We again demonstrate a simulator, this time with $\mathcal{A}$ controlling parties P_1 and P_3. This is similar to the two-party case where P_2 is corrupted. The main challenge is that the simulator needs to construct "fake" garbled circuits in order for $\mathcal{A}$ to output the correct output; however, as shown in the proof of our two-party protocol, such a simulator exists. Thus, the simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 would up until Step 6. $\mathcal{S}$ can extract both P_1's input x and its mask bits $\lambda_{w,j}$ through P_1's calls to $\mathcal{F}^1_{\mathbf{ss}}$ in Step 4 and Step 2. Likewise, in Step 5, $\mathcal{S}$ extracts P_3's input z through the calls to $\mathcal{F}_{\mathbf{ot}}$. $\mathcal{S}$ then passes x and z to the trusted third party, learning $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $\rho \xleftarrow{\$} \{0,1\}^s$. Then for $j \in \{1, \ldots, s\}$, $\mathcal{S}$ proceeds as follows: If $\rho_j = 0$, $\mathcal{S}$ uses the simulator that is known to exist for the two-party circuit garbling protocol to construct garbled circuits which output $f(x, y, z)$. Otherwise, $\mathcal{S}$ acts as P_2 would. $\mathcal{S}$ continues to act as P_2 would, except that in Step 7 it sets the output of $\mathcal{F}_{\mathbf{cf}}$ to be equal to the ρ chosen above. Finally, $\mathcal{S}$ halts, outputting whatever $\mathcal{A}$ outputs.

The main intuition here is that, since P_1 learns nothing about the portion of the circuit garbled by P_2, this reduces to the two-party setting where P_2 is corrupt. Recall that by the security of our garbling protocol, P_1 can only construct circuits that cause the evaluator to abort. If P_1 tries to cheat in Step 5 by exchanging invalid mask shares, P_2 will detect this with high probability, and likewise for Step 8.

The adversary corrupts parties P_2 and P_3. The analysis here is very similar to the case where parties P_1 and P_3 are corrupt.

The adversary corrupts party P_1. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_1 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 and P_3 would, extracting P_1's input x through the calls to $\mathcal{F}^1_{\mathbf{ss}}$ in Step 4. $\mathcal{S}$ passes x to the trusted third party, learning $f(x, y, z)$, and halts, outputting whatever $\mathcal{A}$ outputs.

As $\mathcal{S}$ acts exactly as P_2 and P_3 would, and $\mathcal{A}$ gets no output, we need only show that the probability that $\mathcal{A}$ aborts in both the real and ideal world is identical. In fact, this follows from

our maliciously secure two-party protocol and the security of the input-consistency checks.

The adversary corrupts party P_2. The analysis here is very similar to the case where party P_1 is corrupt.

The adversary corrupts party P_3. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_3 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_1 and P_2 would, extracting P_3's input z through the $\mathcal{F}_{\mathbf{ot}}$ calls in Step 5. $\mathcal{S}$ then hands z to the trusted third party, who returns $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $\rho \xleftarrow{\$} \{0,1\}^s$, and then for $j \in \{1, \ldots, s\}$ $\mathcal{S}$ proceeds as follows: If $\rho_j = 0$, $\mathcal{S}$ constructs a distributed garbled circuit which outputs $f(x, y, z)$, otherwise $\mathcal{S}$ proceeds as normal. Then, in Step 7, $\mathcal{S}$ fixes the output of $\mathcal{F}_{\mathbf{cf}}$ to be ρ. For the rest of the protocol, $\mathcal{S}$ acts as P_1 and P_2 would, and eventually halts, outputting what $\mathcal{A}$ outputs.

The analysis is very similar to [LP11, pp. 22–23]. □

F Proof of Theorem 2

Theorem. *Let C be an arbitrary polynomial-size circuit and let $\mathbb{G}$ be a cyclic group with prime order. Given access to ideal functionalities $\mathcal{F}_{\mathbf{const}}$, $\mathcal{F}_{\mathbf{gate}}$, $\mathcal{F}_{\mathbf{oshare}}$, $\mathcal{F}_{\mathbf{ot}}$, $\mathcal{F}_{\mathbf{rand}}$, and $\mathcal{F}_{\mathbf{ss}}$, and assuming that the decisional Diffie-Hellman problem is hard in $\mathbb{G}$, then $\Pi_{\mathbf{3PC}}^{m-lindell}(P_1, P_2, P_3)$ securely computes the circuit C in the presence of an adversary corrupting an arbitrary number of parties.*

Proof (Sketch). The analysis here is nearly identical to the proof in Appendix E as well as the proof for the two-party case [Lin13], and thus we just present the simulators for each corruption case.

The adversary corrupts parties P_1 **and** P_2. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and the runs as P_3 would, using input $0^{\mathsf{n}-\mathsf{n}_2}$, until Step 11. Here, $\mathcal{S}$ uses the witnesses a^i_{w,b_w} send by P_1 and P_2 for the zero-knowledge proof-of-knowledge to extract their inputs. $\mathcal{S}$ then feeds these inputs to the trusted third party, receiving back $f(x, y, z)$. $\mathcal{S}$ continues to run as P_3 would, and halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts parties P_1 **and** P_3. The simulator $\mathcal{S}$ is constructed as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 would up until Step 6. $\mathcal{S}$ can extract both P_1's input x and its mask bits $\lambda_{w,j}$ through P_1's calls to $\mathcal{F}^1_{\mathbf{ss}}$ in Step 2 and Step 4. Likewise, in Step 5, $\mathcal{S}$ extracts P_3's input z through the calls to $\mathcal{F}_{\mathbf{ot}}$. $\mathcal{S}$ then passes x and z to the trusted third party, learning $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $\rho \xleftarrow{\$} \{0,1\}^s$. Then for $j \in \{1, \ldots, s\}$, $\mathcal{S}$ proceeds as follows: If $\rho_j = 0$, $\mathcal{S}$ uses the simulator that is known to exist for the two-party circuit garbling protocol to construct garbled circuits which output $f(x, y, z)$. Otherwise, $\mathcal{S}$ acts as P_2 would. $\mathcal{S}$ continues to act as P_2 would, except that in Step 7 it sets the output of $\mathcal{F}_{\mathbf{cf}}$ to be equal to the ρ chosen above. Finally, $\mathcal{S}$ halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts parties P_2 **and** P_3. The analysis here is exactly the same as the case where parties P_1 and P_3 are corrupt.

The adversary corrupts party P_1. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_1 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_2 and P_3 would, extracting P_1's input x through the calls to $\mathcal{F}^1_{\mathbf{ss}}$ in Step 4. $\mathcal{S}$ passes x to the trusted third party, learning $f(x, y, z)$. For the rest of the protocol, $\mathcal{S}$ acts as P_2 and P_3 would, and eventually halts, outputting whatever $\mathcal{A}$ outputs.

The adversary corrupts party P_2. The analysis here is exactly the same as the case where party P_1 is corrupt.

The adversary corrupts party P_3. We construct a simulator $\mathcal{S}$ with access to an adversary $\mathcal{A}$ controlling P_3 as follows: $\mathcal{S}$ invokes the adversary $\mathcal{A}$, and runs as P_1 and P_2 would, extracting P_3's input z through the $\mathcal{F}_{\mathbf{ot}}$ calls in Step 5. $\mathcal{S}$ then hands z to the trusted third party, who returns $f(x, y, z)$. In Step 6, $\mathcal{S}$ chooses $\rho \xleftarrow{\$} \{0,1\}^s$, and then for $j \in \{1, \ldots, s\}$ $\mathcal{S}$ proceeds as follows: If $\rho_j = 0$, $\mathcal{S}$ constructs a distributed garbled circuit which outputs $f(x, y, z)$, otherwise $\mathcal{S}$ proceeds as normal. Then, in Step 7, $\mathcal{S}$ fixes the output of $\mathcal{F}_{\mathbf{cf}}$ to be ρ. For the rest of the protocol, $\mathcal{S}$ acts as P_1 and P_2 would, and eventually halts, outputting whatever $\mathcal{A}$ outputs. □

G Concrete Efficiency of the 3PC Protocol

For simplicity we assume each party's input has length ℓ. Since we apply the XOR-tree technique to P_3's input, we let $\ell' = \max\{4\ell, 8s\}$ be the new input length. Finally, we redefine s in this section to be the statistical security parameter; namely, the adversary can succeed in cheating with probability at most 2^{-s}.

Protocol based on [LP07, LP11]. Table 1 details the specific computational cost of each step for $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$. Note that these numbers are across *all* parties; the actual per-party cost is less. Each of the hybrid calls can be instantiated efficiently using known techniques: $\mathcal{F}_{\mathbf{cf}}$ can be instantiated very efficiently in the Random Oracle Model requiring only three oracle calls, $\mathcal{F}_{\mathbf{zkpok}}$ can be instantiated using $3s/2+18$ exponentiations [LP11, pg. 36], and $\mathcal{F}_{\mathbf{ot}}$ can be computed using 3 exponentiations [PVW08].

Step	Exponentiations	Hybrids Calls	Symmetric Ops
1			
2	$4\ell + 2(3s)$	$(2\ell \cdot \mathcal{F}_{\mathbf{ss}} + (\ell' + \mathsf{q}) \cdot \mathcal{F}_{\mathbf{rand}} + 4\ell \cdot H) \cdot (3s)$	
3		$(4\mathsf{q} \cdot \mathcal{F}_{\mathbf{gate}} + 8\mathsf{q} \cdot \mathcal{F}_{\mathbf{oshare}}) \cdot (3s)$	$8(3s)\mathsf{q}$
4		$2\ell \cdot \mathcal{F}_{\mathbf{ss}}$	
5		$(2\ell' \cdot \mathcal{F}_{\mathbf{ot}}) \cdot (3s)$	
6			
7		$\mathcal{F}_{\mathbf{cf}}$	
8			
9	$3s$		$4(3s)\mathsf{q}$
10		$(2\ell \cdot H + 2\ell \cdot \mathcal{F}_{\mathbf{zkpok}}) \cdot (3s)$	
11			$(3s)\mathsf{q}$

Table 1: Computational cost for each step of $\Pi^m_{\mathbf{3PC}}(P_1, P_2, P_3)$.

Protocol based on [Lin13]. The concrete computational cost for this protocol are similar to the ones above, except s in this case is smaller to achieve the same level of security. However, we must run the above protocol as a sub-protocol. See Table 2 for the concrete efficiency counts.

Comparison with SPDZ. We compare our three-party protocol with the SPDZ protocol [BDOZ11, DPSZ12, DKL$^+$12, DKL$^+$13, KSS13], an efficient MPC protocol which works for n parties and arbitrary corruptions over arithmetic circuits, and follows the preprocessing paradigm. SPDZ represents the state-of-the-art at the time of writing in terms of efficiency in the multi-party setting. Here

Step	Exponentiations	Hybrids Calls	Symmetric Ops
1			
2	$4\ell + 2s$	$(2\ell \cdot \mathcal{F}_{\mathbf{ss}} + (\ell' + \mathsf{q})\mathcal{F}_{\mathbf{rand}} + 4\ell \cdot H) \cdot s$	
3		$(4\mathsf{q} \cdot \mathcal{F}_{\mathbf{gate}} + 8\mathsf{q} \cdot \mathcal{F}_{\mathbf{oshare}}) \cdot s$	$8s\mathsf{q}$
4		$2\ell \cdot \mathcal{F}_{\mathbf{ss}}$	
5		$(2\ell' \cdot \mathcal{F}_{\mathbf{ot}}) \cdot s$	
6		$(2\mathsf{m} \cdot H) \cdot s$	
7		$\mathcal{F}_{\mathbf{cf}}$	
8		$((2\ell + \mathsf{m}) \cdot H) \cdot s$	$s\mathsf{q}$
9	— Cost of running $\Pi^{m}_{\mathbf{3PC}}(P_1, P_2, P_3)$ on circuit of size roughly $\mathcal{O}(\ell + \mathsf{m})$ —		
10	s		$4s\mathsf{q}$
11		$(2\ell \cdot \mathcal{F}_{\mathbf{zkpok}}) \cdot s$	
12			

Table 2: Computational cost for each step of $\Pi^{m-lindell}_{\mathbf{3PC}}(P_1, P_2, P_3)$.

we focus on the differences between both SPDZ and our protocol, and discuss their strengths and weaknesses. Due to the different characteristics of each protocol (e.g., arithmetic versus boolean, linear versus constant round, etc.), these protocols are somewhat "incomparable". However, we hope to give a general idea of the efficiency trade-offs of both protocols.

There are several key differences between the SPDZ protocol and our own. For one, SPDZ works over arithmetic circuits, whereas our protocol works over boolean circuits.[6] In terms of communication, the SPDZ protocol requires rounds linear in the depth of the circuit, whereas our protocol is constant-round. While it is difficult to compare the impact of this without an implementation and experiments, it seems intuitive that as the latency between machines increases, the cost of each additional communication round increases as well; this intuition has been backed up by experiments in the semi-honest setting [SZ13].

Finally, we consider the start-to-finish execution time (i.e., including the cost of preprocessing) for running an AES circuit. The preprocessing in our protocol is basically that found in [NNOB12], and, using the numbers presented there, is fairly efficient (around 1 minute [NNOB12, Figure 21]). Efficiency comes from the fact that the preprocessing is only between two parties, namely, the circuit generators. The on-line running time is conjectured to be around that of maliciously secure two-party protocols using cut-and-choose.

The SPDZ protocol, on the other hand, has a very efficient (information-theoretic) online phase but a much costlier offline phase (around 17 minutes for three parties [DKL$^+$12, Table 2]). In addition, it has a one-time setup phase which is very costly: the parties need to execute an MPC protocol for a circuit which generates a key pair with the secret key secret-shared among the parties. Executing this on its own would likely eclipse the running time of our protocol.[7] Thus, given preprocessing, it seems likely that SPDZ would out-perform our protocol; however, in the setting of executing the protocol from start to finish, we conjecture that our protocol would be more efficient.

Finally, our protocol is most efficient in the Random Oracle Model, whereas SPDZ works in the standard model.

[6] One recent work [DZ13] develops a SPDZ-like protocol for boolean circuits, however its practical efficiency is unclear.

[7] We note that [DKL$^+$13] present an efficient protocol for this one-time setup phase in the weaker *covert* security model.

Changelog

– Version 1.0 (February 19, 2014): First release