Efficient Three-Party Computation from Cut-and-Choose

Seung Geol Choi[1⋆], Jonathan Katz[2], Alex J. Malozemoff[2], and Vassilis Zikas[3⋆⋆]

[1] Computer Science Department, United States Naval Academy
choi@usna.edu
[2] Computer Science Department, University of Maryland
{jkatz,amaloz}@cs.umd.edu
[3] Computer Science Department, ETH Zurich
vzikas@inf.ethz.ch

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 only a small constant worse than 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., [19, 23, 24]). With only a few exceptions [13, 14, 23], however, this work has tended to focus on secure *two-party* computation (2PC). In the semi-honest setting, a series of papers [3, 17–19] showed that Yao's garbled circuit technique [38] can yield very efficient protocols for the computation of boolean circuits. In the malicious setting, Lindell and Pinkas [27] 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 [20, 24, 25, 28, 29, 31, 33, 34, 36, 37], 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 [15] (e.g., the TinyOT protocol [32]). Although this approach yields protocols

⋆ Portions of this work were done while at the University of Maryland.
⋆⋆ Portions of this work were done at the University of Maryland and UCLA.

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 optimizing MPC for *semi-honest* adversaries [3, 5–8, 10], less work has focused on security against malicious corruptions. The work of Ishai, Prabhakaran, and Sahai [22] gives protocols with good *asymptotic* efficiency; however, despite some promising optimizations [26], it has not yet produced practical instantiations. The SPDZ protocol [4, 12–14, 23], 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 [12–14] 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 [35] have shown that as the latency between machines increases, the cost of each round becomes more and more significant.

Finally, the work of Goyal, Mohassel, and Smith [16] uses the cut-and-choose technique to construct a multi-party protocol secure in the *covert* setting.

Multi-party computation 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.[4] Yet, in practice, it seems that the most likely scenarios for secure MPC would involve a small number of parties [5]. 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.

In addition, the three-party setting is interesting in its own right. For example, suppose the government would like to run some privacy preserving computation on a company's dataset, such as flight manifests. Now, suppose the public does not trust that these parties are not colluding. Thus, we could add a third party, trusted by the public, into the computation to enforce that the two main parties are not simply sharing all their information.

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 existing (cut-and-choose-based) 2PC

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

protocols 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 eight overhead over the underlying 2PC protocol in terms of computation, and a factor sixteen 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. 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. The most efficient instantiation of our protocol requires the random oracle model.

Overview of our 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.

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 $t \cdot \hat{t}$, yielding at least a quadratic blowup.

Emulating the sender vs. 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 [1, 11] 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.

Our distributed garbling scheme is secure against malicious adversaries, which ensures that an adversary corrupting only one of the parties P_1 or 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 [27, 28]. In the full version [9], we present a construction based on the more recent protocol by Lindell [25] 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.

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 Bellare, Hoang, and Rogaway [2]. A circuit C is defined by parameters $(\mathsf{n}, \mathsf{m}, \mathsf{q}, L, R, G)$, 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 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 can construct such authenticated sharings using the TinyOT protocol [32]; see the full version [9] for details.

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 [11]. 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.

We describe our construction in several steps. 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

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

1. **Generate masks:**
   - For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}-\mathsf{m}\}$: set $\lambda_w \xleftarrow{\$} \{0,1\}$.
   - For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: set $\lambda_w \leftarrow 0$.
2. **Generate sub-keys:**
   - For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}\}$ and $b \in \{0,1\}$: set $s^1_{w,b}, s^2_{w,b} \xleftarrow{\$} \{0,1\}^k$.
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 index of the left and right input wires, respectively, of the gate indexed by γ. Letting $K_{w,b} = (s^1_{w,b}, s^2_{w,b})$, for $i, j \in \{0,1\}^2$, compute the following:

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

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

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

Fig. 1. Circuit garbling scheme.

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 Damgård and Ishai protocol [11] 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.

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 $(K_{\beta, b_\beta \oplus \lambda_\beta}, b_\beta \oplus$

λ_β), 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* $\mathcal{F}^i_{\mathbf{ss}}(b)$: 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 the full version [9] for details.

Distributed encryption scheme. We utilize Damgård and Ishai's distributed encryption scheme [11]. Suppose the message and the key for the encryption scheme are distributed as follows:

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

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

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

where F^1_k is a PRF keyed by key k. To decrypt a ciphertext $c := \mathsf{Enc}^T_K(m)$, 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 $K_\alpha = (s^1_\alpha, s^2_\alpha)$ and $K_\beta = (s^1_\beta, s^2_\beta)$, where P_i holds (s^i_α, s^i_β), encryption with tweak T works as follows:

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

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 details. As before, for each wire w in the circuit we associate keys $K_{w,0} = (s^1_{w,0}, s^2_{w,0})$ and $K_{w,1} = (s^1_{w,1}, s^2_{w,1})$ 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 $(s^i_{w,0}, s^i_{w,1})$. 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 $\langle\sigma_{\gamma,b_\alpha,b_\beta}\rangle = \mathcal{F}^G_{\mathbf{gate}}(\langle b_\alpha\rangle \oplus \langle\lambda_\alpha\rangle, \langle b_\beta\rangle \oplus \langle\lambda_\beta\rangle) \oplus \langle\lambda_\gamma\rangle$. 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 $\mathcal{F}^i_{\mathbf{oshare}}(\langle\sigma_{\gamma,b_\alpha,b_\beta}\rangle, s^i_{\gamma,0}, s^i_{\gamma,1})$. 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 the full version [9] 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.

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:

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

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

1. **Generate mask bits:**
   - For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 sets $\lambda_w \xleftarrow{\$} \{0,1\}$ and $\langle \lambda_w \rangle \leftarrow \mathcal{F}^1_{\mathbf{ss}}(\lambda_w)$.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}\}$: P_2 sets $\lambda_w \xleftarrow{\$} \{0,1\}$ and $\langle \lambda_w \rangle \leftarrow \mathcal{F}^2_{\mathbf{ss}}(\lambda_w)$.
   - For $w \in \{\mathsf{n}+1, \ldots, \mathsf{n}+\mathsf{q}-\mathsf{m}\}$: set $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\mathbf{rand}}$.
   - For $w \in \{\mathsf{n}+\mathsf{q}-\mathsf{m}+1, \ldots, \mathsf{n}+\mathsf{q}\}$: set $\langle \lambda_w \rangle \leftarrow \mathcal{F}^0_{\mathbf{const}}$.
2. **Generate sub-keys:**
   - For $w \in \{1, \ldots, \mathsf{n}+\mathsf{q}\}$ and $b \in \{0,1\}$: P_i sets $s^i_{w,b} \xleftarrow{\$} \{0,1\}^k$.
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. For $i, j \in \{0,1\}^2$, compute the following selector bits:

$$\langle \sigma_{\gamma,i,j} \rangle \leftarrow \mathcal{F}^{G_\gamma}_{\mathbf{gate}}(\langle \lambda_\alpha \rangle \oplus \langle i \rangle, \langle \lambda_\beta \rangle \oplus \langle j \rangle) \oplus \langle \lambda_\gamma \rangle.$$

     Next, for $i, j \in \{0,1\}^2$, compute sharings of the appropriate sub-keys to use for each row:

$$\begin{bmatrix} \hat{s}^1_{\gamma,i,j} \end{bmatrix} \leftarrow \mathcal{F}^1_{\mathbf{oshare}}(\langle \sigma_{\gamma,i,j} \rangle, s^1_{\gamma,0}, s^1_{\gamma,1}),$$
$$\begin{bmatrix} \hat{s}^2_{\gamma,i,j} \end{bmatrix} \leftarrow \mathcal{F}^2_{\mathbf{oshare}}(\langle \sigma_{\gamma,i,j} \rangle, s^2_{\gamma,0}, s^2_{\gamma,1}).$$

     Finally, for $i, j \in \{0,1\}^2$, 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:

$$(P^1[\gamma,i,j], P^2[\gamma,i,j]) \leftarrow \mathsf{Enc}^{\gamma \| i \| j}_{K_{\alpha,i}, K_{\beta,j}}(\left[\hat{s}^1_{\gamma,i,j}\right] \| \left[\hat{s}^2_{\gamma,i,j}\right] \| \langle \sigma_{\gamma,i,j} \rangle).$$

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

Fig. 2. Two-party distributed circuit-garbling protocol $\Pi_{\mathbf{GC}}(P_1, P_2)$. For semi-honest security, use standard secret sharing; for malicious security use authenticated secret sharing.

- $\mathcal{F}^1_{\mathbf{const}}()$ 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.

Observe that we only authenticate sharings of bits and *not* sharings of the sub-keys $s^i_{w,b}$. This complicates the proof, as the sharing does not provide means of protecting against a malicious party sending inconsistent key-shares, but yields a more efficient construction; see the full version [9] for details.

We also need 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

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

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 the full version [9] 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 [27, 28] to the three-party setting. In the full version [9] we use this "generic" protocol to show how to adapt Lindell's protocol [25] (the current state-of-the-art garbled-circuit-based protocol at the time of writing) to the three-party setting. The cost of each of these three-party protocols is roughly *eight times* the computational cost of the underlying two-party protocol they are based on, and roughly *sixteen times* the communication cost (plus the cost of a small number of OTs per gate, which can be efficiently amortized using OT extension [21, 32]), 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; see the full version [9] for the details.

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 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 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 [27, 28] 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 the full version [9] we show how we can use this protocol in an adaptation of Lindell's protocol [25] 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 [30, 28] and adapt it in a straightforward manner to the three-party setting.
– *Selective failure.* This attack arises when the parties execute OT to send the sub-keys for P_3's input. Note that if the sender in the OT 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 [27].

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 the protocol of Lindell and Pinkas [28].
3. P_1 and P_2 construct s garbled circuits using $\Pi'_{\mathbf{GC}}$ and the input sub-keys generated as in the protocol of Lindell and Pinkas [28].
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 (and likewise for P_2). 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 a zero-knowledge proof-of-knowledge protocol [28], 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 sets $a^1_{w,0}, a^1_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs $\left\{(w, 0, g^{a^1_{w,0}}), (w, 1, g^{a^1_{w,1}})\right\}$.

   For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 sets $a^2_{w,0}, a^2_{w,1} \xleftarrow{\$} \mathbb{Z}_q$ and constructs $\left\{(w, 0, g^{a^2_{w,0}}), (w, 1, g^{a^2_{w,1}})\right\}$.

   For $j \in \{1, \ldots, s\}$: P_i, for $i \in \{1, 2\}$, sets $r^i_j \xleftarrow{\$} \mathbb{Z}_q$ and constructs $\left\{(j, g^{r^i_j})\right\}$.

   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 sets $s^1_{w, b \oplus \lambda_{w,j}, j} \leftarrow H(g^{a^1_{w,b} \cdot r^1_j})$ for $b \in \{0, 1\}$.
   - For $w \in \{\mathsf{n}_1 + 1, \ldots, \mathsf{n}_2\}$: P_2 sets $s^2_{w, b \oplus \lambda_{w,j}, j} \leftarrow H(g^{a^2_{w,b} \cdot r^2_j})$ 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 from Step 2, along with $\left\{GC^i_j\right\}_{i=1}^s$, to P_3.
7. The parties set $\rho \leftarrow \mathcal{F}_{\mathbf{cf}}$. Let $\mathcal{CC} = \{i : \rho_i = 1\}$, and $\mathcal{EC} = \{1, \ldots, s\} \setminus \mathcal{CC}$.
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 sets $s^1_{w,b_w \oplus \lambda_{w,j},j} \leftarrow H(g^{a^1_{w,b_w} \cdot r^1_j})$.
    - 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 sets $s^2_{w,b_w \oplus \lambda_{w,j},j} \leftarrow H(g^{a^2_{w,b_w} \cdot r^2_j})$.

    For $w \in \{1, \ldots, \mathsf{n}_1\}$: P_1 and P_3 run $\mathcal{F}_{\mathbf{zkpok}}$, with P_1 acting as the prover inputting a^1_{w,b_w} and P_3 acting 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 acting as the prover inputting a^2_{w,b_w} and P_3 acting 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 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.

In the full version [9] we prove the following.

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.*

4.3 Efficiency

We now argue why our 3PC protocol is roughly eight times as expensive in terms of computation as the underlying 2PC protocol we utilize, and roughly sixteen times as expensive in terms of communication. Both protocols are very similar to the underlying 2PC protocol they are based on; the major changes in terms of computational cost are that (1) the cost of encrypting a single row increases due to the use of the distributed encryption scheme, and (2) P_3 needs to do twice the work (due to communicating with *both* P_1 and P_2) as compared to the evaluator in the underlying 2PC protocol. Indeed, it takes about eight PRF calls (where one PRF call equals outputting k bits) to encrypt a single row of the garbled circuit, and thus the cost and size of a garbled circuit increases by a factor of eight. 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 [21].

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. Since each garbled circuit is eight times larger than in the underlying 2PC protocol, we find that the overall communication size increases my approximately sixteen.

Comparison with SPDZ. We compare our three-party protocol with the SPDZ protocol [4, 12–14, 23], an efficient protocol over arithmetic circuits that works for n parties and arbitrary corruptions, and uses the preprocessing paradigm. SPDZ represents the state-of-the-art in terms of efficiency in the multi-party setting. Here 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. 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 [35]. And while SPDZ works in the standard model, the most efficient instantiation of our protocol requires the random oracle model.

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 the TinyOT protocol [32], and, using the numbers presented there, is fairly efficient (around 1 minute [32, 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 [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.[5] 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.

Acknowledgments

Work of Seung Geol Choi supported in part by the Office of Naval Research under Grant Number N0001414WX20588. Work of Jonathan Katz supported in part by NSF awards #0964541 and #1111599. Work of Alex J. Malozemoff supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFG 168a, awarded by DoD, Air Force Office of Scientific Research. Work of Vassilis Zikas supported in part by NSF awards #09165174, #1065276, #1118126, and #1136174, US-Israel BSF grant #2008411, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Corporation Research Award, the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392, and Swiss National Science Foundation (SNF) Ambizione grant PZ00P2_142549. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

1. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: 22nd ACM STOC. pp. 503–513. ACM Press (1990)
2. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 12. pp. 784–796. ACM Press (2012)
3. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: Ning, P., Syverson, P.F., Jha, S. (eds.) ACM CCS 08. pp. 257–266. ACM Press (2008)
4. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer (May 2011)
5. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer (Oct 2008)

[5] We note that Damgård et al. [13] present an efficient protocol for this one-time setup phase in the weaker *covert* security model.

6. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer (Feb 2009)
7. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: Goldberg, I. (ed.) 19th USENIX Security Symposium. USENIX Association, Washington, D.C., USA (Aug 2010)
8. Choi, S.G., Hwang, K.W., Katz, J., Malkin, T., Rubenstein, D.: Secure multi-party computation of Boolean circuits with applications to privacy in on-line marketplaces. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 416–432. Springer (Feb / Mar 2012)
9. Choi, S.G., Katz, J., Malozemoff, A.J., Zikas, V.: Efficient three-party computation from cut-and-choose. Cryptology ePrint Archive, Report 2014/128 (2014), https://eprint.iacr.org/
10. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 160–179. Springer (Mar 2009)
11. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 378–394. Springer (Aug 2005)
12. Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.P.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Visconti, I., Prisco, R.D. (eds.) SCN 12. LNCS, vol. 7485, pp. 241–263. Springer (Sep 2012)
13. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer (Sep 2013)
14. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer (Aug 2012)
15. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987)
16. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 289–306. Springer (Apr 2008)
17. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: Tool for automating secure two-party computations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) ACM CCS 10. pp. 451–462. ACM Press (Oct 2010)
18. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS 2012. The Internet Society (Feb 2012)
19. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Wagner, D. (ed.) 20th USENIX Security Symposium. USENIX Association, San Francisco, California, USA (Aug 2011)
20. Huang, Y., Katz, J., Evans, D.: Efficient secure two-party computation using symmetric cut-and-choose. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 18–35. Springer (Aug 2013)
21. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer (2003)

22. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer - efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer (Aug 2008)
23. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 13. pp. 549–560. ACM Press (Nov 2013)
24. Kreuter, B., Shelat, A., Shen, C.H.: Towards billion-gate secure computation with malicious adversaries. In: Kohno, T. (ed.) 21st USENIX Security Symposium. USENIX Association, Bellevue, Washington, USA (Aug 2012)
25. Lindell, Y.: Fast cut-and-choose based protocols for malicious and covert adversaries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 1–17. Springer (Aug 2013)
26. Lindell, Y., Oxman, E., Pinkas, B.: The IPS compiler: Optimizations, variants and concrete efficiency. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 259–276. Springer (Aug 2011)
27. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer (May 2007)
28. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 329–346. Springer (2011)
29. Lindell, Y., Pinkas, B., Smart, N.P.: Implementing two-party computation efficiently with security against malicious adversaries. In: Ostrovsky, R., Prisco, R.D., Visconti, I. (eds.) SCN 08. LNCS, vol. 5229, pp. 2–20. Springer (Sep 2008)
30. Mohassel, P., Franklin, M.: Efficiency tradeoffs for malicious two-party computation. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 458–473. Springer (Apr 2006)
31. Mohassel, P., Riva, B.: Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 36–53. Springer (Aug 2013)
32. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer (Aug 2012)
33. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer (Mar 2009)
34. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer (Dec 2009)
35. Schneider, T., Zohner, M.: GMW vs. Yao? efficient secure two-party computation with low depth circuits. In: Sadeghi, A.R. (ed.) FC 2013. LNCS, vol. 7859, pp. 275–292. Springer (Apr 2013)
36. Shelat, A., Shen, C.H.: Two-output secure computation with malicious adversaries. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 386–405. Springer (May 2011)
37. Shelat, A., Shen, C.H.: Fast two-party secure computation with minimal assumptions. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 13. pp. 523–534. ACM Press (Nov 2013)
38. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press (Oct 1986)