Secure Computation with Low Communication from Cross-checking

S. Dov Gordon
George Mason University
gordon@gmu.edu

Samuel Ranellucci
University of Maryland
George Mason University
samuel@umd.edu

Xiao Wang
University of Maryland
wangxiao@cs.umd.edu

Abstract

We construct new four-party protocols for secure computation that are secure against a single malicious corruption. Our protocols can perform computations over a binary ring, and require sending just 1.5 ring elements per party, per gate. In the special case of Boolean circuits, this amounts to sending 1.5 bits per party, per gate. One of our protocols is robust, yet requires almost no additional communication. Our key technique can be viewed as a variant of the "dual execution" approach, but, because we rely on four parties instead of two, we can avoid any leakage, achieving the standard notion of security.

1 Introduction

As secure multi-party computation (MPC) is transitioning to practice, one setting that has motivated multiple deployments is that of outsourced computation, in which hundreds of thousands, or millions of users secret share their input among some small number of computational servers. In this setting, the datasets can be extremely large, while the number of computing parties is small. The use of secure computation in such settings is often viewed as a safeguard that helps to reduce risk and liability. While companies and government agencies are increasingly choosing to deploy this safeguard, it is a security / performance tradeoff that many are not yet willing to make.

One important notion related to the security of an MPC protocol is the choice of adversarial threshold: a higher threshold means that the protocol can tolerate more corrupted parties, and is thus more secure. However, requiring a higher threshold usually results in feasibility and efficiency obstacles. For example, the earliest results in the field demonstrated key distinctions between $t \geq n/2$, $t < n/2$, and $t < n/3$ corruptions [RBO89, BMR90, GMW87, BOGW88], including whether fairness could be guaranteed ($t < n/2$), whether a broadcast channel is needed ($t > n/3$), and whether cryptographic assumptions are necessary ($t > n/3$). More recently, when $t > n/2$, there are results showing how to reduce the bandwidth to just a constant number of field elements per party, per gate [FY92, DIK$^+$08, DIK10]. In contrast, when $t \geq n/2$, our best protocols require expensive preprocessing, with communication cost that grows quadratically in n.

In this work, we develop a new protocol in the honest majority setting, tailored to the case where $n = 4$. Our protocol is secure against a single malicious corruption, consistent with the requirement that $t < n/2$. Focusing on this domain, we are able to construct extremely efficient protocols.

Looking at concrete costs, the most efficient secure two-party computation protocol (in terms of communication) requires roughly 290 bytes of communication per party per gate [WRK17a,

NNOB12]. If we are willing to relax the setting by assuming that a malicious adversary can only corrupt one out of three parties, then we can further reduce the cost to 7 bits per party per gate [ABF+17]. Our protocol further reduces the cost significantly: our four-party protocol requires only 1.5-bits of communication per party. Furthermore, the results just cited for the two-party and three-party settings are for 40-bit statistical security, and their costs per gate increase for higher statistical security. Our protocol has no dependence on a statistical security parameter, and has only an additive $O(\kappa)$ term (where κ is a computational security parameter).

We also note that we can achieve 1-bit communication per party in the six-party setting. For these previous works as well as the protocol in this paper, all computation can be hardware accelerated and thus communication complexity is the most suitable indicator of real performance.

Contributions. We now summarize our contributions. Our main result is summarized in the theorem below. The construction and proof of security appear in Sections 3 and 4. An additional improvement appears in Section 5.

Theorem 1. *In the four party setting, it is possible to construct a protocol for securely computing a circuit of size $|C|$ whose total communication complexity is $6|C|\log|F| + O(\kappa)$. In particular, for a Boolean circuit, this amounts to 1.5 bits per player, per gate.*

Binary Rings. An interesting result of our work is that we can securely evaluate an arithmetic function over binary rings, such as $(\mathbb{Z}_{2^{32}}, +, *)$, where $(+, *)$ denotes modular addition and multiplication. Note that most MPC protocols do not work over rings that are not fields. In particular, MAC-based protocols based on SPDZ [DPSZ12] do not work over $\mathbb{Z}_{2^{32}}$, as the multiplicative inverse is necessary for constructing linear MAC schemes. The security of our protocol only relies on additive maskings, so we do not need a multiplicative inverse. The correctness of our protocol when computing over a binary ring follows from the distributivity property of rings. A similar observation, in the semi-honest setting, was recently made by Mohassel and Zhang [MZ17].

Robustness. We construct a robust variant of our protocol, guaranteeing that the honest parties always receive correct output. The cost of adding robustness is free if nothing goes wrong, and requires an additional $O(\kappa \log|C| \log|\mathbb{F}|)$ overhead when a player misbehaves (Section 6).

1.1 Technical Overview

From a high-level view, the construction of our protocol starts with a semi-honest protocol, π_1, for two-party computation in the preprocessing model. We would like two participants in the protocol to execute π_1. There are two main tasks towards our final goal:

1. Generating the preprocessing data for π_1 with malicious security.
2. Strengthening the security of π_1 from semi-honest to malicious security.

Our solutions to these challenges rely heavily on the fact that we work in the four-party setting with only one corruption. In order to generate maliciously secure preprocessing, we ask the other two parties to locally emulate the preprocessing ideal functionality, both using the same randomness. To ensure that the computation of the preprocessing is done correctly, each of the parties executing π_1 verifies that he was sent two identical copies of the preprocessing.

The second challenge is trickier. Existing work that compiles semi-honest security to malicious security are not suitable for our use. The techniques can be broadly described as follows: 1) Using

generic zero-knowledge proof, which is impractical for most cases; 2) Using certain forms of MACs on each parties' share to ensure honest behavior. This approach has been made practical, but it requires preprocessing data of size (at least) $\Omega(\rho)$ bits per gate, to achieve $2^{-\rho}$ statistical security. 3) In the honest majority setting, one can use Shamir secret sharing, but our π_1 is a two-party protocol, where one can be malicious. Instead, our approach is based on a technique called "dual-execution" [MF06, HKE12], which is known to have one-bit leakage in general. However, we show that in the four-party setting, by performing a special cross checking protocol at the end, we are able to eliminate the leakage without any penalty to the performance. Details follow below.

Dual execution without leakage. In order to accommodate dual execution, we require that π_1 has certain special properties. Intuitively, the outcome of π_1 should leave both parties with "masked wire values" for all wires in the circuit, together with a secret sharing of the masks. This property can be satisfied by many protocols, e.g. the modified Beaver triple protocol [Bea92] as we used in the paper, as well as the semi-honest version of TinyTable [DNNR17].

Now we are in the setting, where, say, P_1 and P_2 have generated the preprocessing, and each hold the full set of wire masks, namely λ^1. P_3 and P_4 have executed π_1, and recovered masked values, namely m^1. Our dual execution is done by letting P_1 and P_3 switch roles with P_2 and P_4. As a result, P_1 and P_2 will obtain m^2, while P_3 and P_4 will obtain λ^2 in the second execution. Conceptually, our cross-checking compares, for all wire values in the circuit, whether

$$\lambda^1 + \mathsf{m}^2 = \lambda^2 + \mathsf{m}^1.$$

Note that the above holds if both executions are honest, since both sides of the equation are equal to the true wire values, masked by both masks (λ^1 and λ^2). For details of the protocol, see Section 3.

Readers that are familiar with the dual execution paradigm in the two-party setting, from garbled circuits, might wonder how we remove the bit of leakage. There are two key insights here. First, when using garbled circuits, it seems difficult to check the consistency of internal wires, whereas the masked wires of the form just described allows us to easily check the consistency of *all* wires in the two evaluations. This eliminate the possibility of input inconsistency, and also prevents the adversary from flipping a wire value to see if it has any impact on the output. Second, in a garbled circuit implementation, the adversary can fix the output of a particular gate arbitrarily, creating a "selective failure attack": the change goes undetected if the output he chooses is consistent with the true output on that wire, and would otherwise cause an abort. With these masked wire evaluations, the adversary cannot fix a wire value arbitrarily; he is limited to adding some value to the wire, changing it in all cases, and always causing an abort. In particular, then, whether he is caught cheating no longer depends on any private value. By exploiting the structure of masks and masked values, checking for inconsistencies requires only $O(\kappa)$ bits of communication.

Reducing communication. The protocol described until this point is already extremely efficient, but we further reduce the communication in several interesting ways. In the preprocessing, we do this in a fairly straightforward way, using PRG seeds and hash functions to compress the material. In the cross checking, recall that we need the parties to verify, twice, whether $\lambda^1 + \mathsf{m}^2 = \lambda^2 + \mathsf{m}^1$, where these values have size $|C|$. (They verify twice because each member of one evaluation compares with one member of the other evaluation.) A naive way here would be to twice compare the hash of these values, but this is in fact insecure. If an adversary changes a value on one of the wires in his evaluation, as we have already noted, he will always be caught, because his partner will compare the hash of his modified masked wire values with an honest party from the other

evaluation. However, the adversary can still learn sensitive information from the result of his *own* comparison with a member of the other evaluation. Instead, we can use any honest-majority, four-party protocol for comparing these two hash values. The circuit for this comparison has only $O(\kappa)$ gates, so this introduces very little overhead. Nevertheless, in Section 5 we show how to bootstrap this comparison, removing the reliance on other protocols.

Related work. Maliciously secure protocols, tailored for the three-party setting, have been studied in many works. Choi et al. [CKMZ14] studied the dishonest majority setting based on garbled circuits. Araki et al. [ABF+17], Mohassel et al. [MRZ15], Furukawa et al. [FLNW17] studied the honest majority setting. However, we are not aware of any MPC protocol tailored for the four-party setting.

Other protocols that work in the four-party setting include honest majority protocols [BDNP08, DGKN09, DI05, DI06, LN17] and dishonest majority protocols [IPS08, BDOZ11, DPSZ12, NNOB12, LPSY15, WRK17b]. These protocols can be used for MPC with more parties, but when applied in the four-party setting, their concrete performances are worse than our protocol.

2 Preliminaries

In this paper, we mainly consider arithmetic circuits C with addition gates and multiplication gates. Each gate in the circuit is represented as (a, b, c, T), where $T \in \{+, \times\}$ is the operation; a and b are the input wire indices; and c is the output wire index.

We denote the set of wires as $\mathcal{W}$, the set of input wires as $\mathcal{W}_{\texttt{input}}$, the set of output wires of all addition gates as $\mathcal{W}_{\texttt{plus}}$, the set of output wires of all multiplication gates as $\mathcal{W}_{\texttt{mult}}$.

Masked evaluation. One important concept that we use in the paper is masked evaluation. Intuitively, every wire w in the circuit, including each input and output wire, is associated with a random mask, namely λ_w. The masked evaluation procedure works in a way such that for each gate two parties, holding masked input and some helper information, are able to obtain the masked output. All parties hold only secret shares of λ_w, namely $\langle \lambda_w \rangle$, therefore obtaining masked wire values does not reveal any information. We will use m_w to denote the masked wire value on wire w. That is, $\mathsf{m}_w = \lambda_w + x$, assuming that the underlying wire value on wire w is x.

Secure evaluation-masking two-party protocol. A secure two-party protocol for computing circuit C is an evaluation-masking scheme if (1) the protocol uses preprocessing, (2) the preprocessing assigns to the circuit C a masking λ (3) the players evaluate the gates of the circuit layer by layer; if a gate g is in layer L, then the evaluation of L allows both players to learn the masked values for the given layer, (4) if an adversary starts deviating from the protocol, the adversary should not learn any information about the computation unless the output is revealed. (5) any misbehavior from the adversary for a given wire is equivalent to him adding a fixed value to the wire that can be computed from his misbehavior. This type of attack is described as an additive attack in the work of [GIP+14]. They showed that certain MPC protocols have this property.

In this paper, we build upon a variant of Beaver's scheme [Bea92] which is an evaluation-masking scheme. The main modification of beaver is that the players will hold for each wire, secret shares of masks and both players will learn the sum of the mask and the actual underlying value. We denote the sum of a mask and a value as either a masking or a masked value.

Committing encryption. A public-key encryption scheme is committing if the ciphertexts serve as commitments.

- **Completness** A person who encrypts a message m resulting in a ciphertext c needs to be able to prove that c is indeed an encryption of m.
- **Soundness** If the player who generated c can prove that c is an encryption of m then $\mathsf{dec}(sk, c) = m$.
- **Verifiability** Given the public-key, it is easy to determine if a ciphertext is valid.

Theorem 2. *(Informal) From any secure evaluation-masking two-party protocol π_1, secure against a semi-honest adversary, we can construct a protocol π_2 for four parties that is secure against a malicious adversary corrupting at most one player.*

3 Our Main Construction

A quick summary of our idea is that we run two executions of a two-party, semi-honest protocol in the preprocessing model, and verify consistency between these two executions through a strategy that we call cross-checking. We start by partitioning the players into two evaluation groups with two players in each group. Each group prepares preprocessing for the other. They leverage the fact that there is at most one corruption to verify that the preprocessing was done correctly. Then, each group evaluates the circuit using that preprocessing. As the outcome of the evaluation, each party holds masked wire values for all wires in the circuit. Finally, the two groups check the consistency of the two evaluations using their masked wire values and masks. Since one of the evaluations is guaranteed to be correct, any cheating will be caught in this step. Below we provide the details of each of these steps as well as why it is secure. A formal description of the protocol appears in Fig. 1, and in the other figures referenced from there.

Pre-processing. Recall that we partition four parties into two equal-sized groups. We first let one group create preprocessing material, and distribute the preprocessing to the other group. This procedure is then repeated with the roles reversed; we describe it only for one group. We will often refer to the group that is performing the pre-processing step as D_1 and D_2, and to the group that uses the preprocessing in the evaluation phase as E_1 and E_2, recognizing that one party plays the role of (say) D_1 in one execution while playing E_1 in the other execution. An ideal functionality for the pre-processing appears in Fig. 2.

To generate the preprocessing material, D_2 chooses a random string and sends it to D_1. They then each use this randomness to locally generate preprocessing, choosing mask values for every wire in the circuit as follows. They select a random field element for every wire $w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}$ (that is, for every input wire, and every wire that is the output of a multiplication gate). We refer to these mask values as λ^1, and the ones generated by the other 2 parties, in the second preprocessing execution, are denoted by λ^2. For the output wire of addition gate $(a, b, c, +)$, suppose the input wires a and b have already been assigned mask values λ_a and λ_b. Then the output wire of the gate is assigned the mask value $\lambda_a + \lambda_b$. Note that all circuit wires now have well defined masks. For each multiplication gate $(a, b, c, \times)$, the two parties additionally compute $\gamma_c = \lambda_a \cdot \lambda_b$. We let $\gamma^1 = \{\gamma_c\}_{c \in \mathcal{W}_{\mathtt{mult}}}$. D_1 and D_2 use their shared random string to construct secret sharings $\lambda^1 = \Lambda_1 + \Lambda_2$ and $\gamma^1 = \Gamma_1 + \Gamma_2$. That is, they create two identical copies of the secret sharing.

The main protocol takes input from all 4 parties, and outputs the evaluation of C on those inputs. It makes use of the 3 components: $\mathcal{F}_{\mathsf{pre}}$, π_{eval}, and π_{cross}.

Pre-processing

1. The four parties make two calls to $\mathcal{F}_{\mathsf{pre}}$ (Fig. 2). In the first call, P_1 and P_2 receive the output $\langle\lambda^2\rangle, \langle\gamma^2\rangle$ for E_1 and E_2 respectively, while P_3 and P_4 receive λ^2, the output of D_1, D_2. In the second call, they reverse their roles, with P_1 and P_2 receiving λ^1, the output of D_1 and D_2, and P_3 and P_4 receive $\langle\lambda^1\rangle, \langle\gamma^1\rangle$, the output of E_1 and E_2.

Evaluation

1. The four parties run two instances of π_{eval} (Fig. 4). In the first instance, the players P_1 and P_2 take the role of evaluators E_1 and E_2 using $\langle\lambda^2\rangle, \langle\gamma^2\rangle$. Let m^1 denote the resulting masked wire values. In the second instance P_3 and P_4 take the role of evaluators E_1 and E_2 using $\langle\lambda^1\rangle, \langle\gamma^1\rangle$. Let m^2 denote the resulting masked evaluation.

Cross Checking

1. The four parties run π_{cross} (Fig. 5) where P_1, P_2 each input m^1, λ^2 while P_3, P_4 both input m^2, λ^1.
2. If π_{cross} outputs 0, then abort.
3. We define $\lambda^1_{\mathtt{out}}$ to be output masks for the first evaluation.
4. We define $\mathsf{m}^1_{\mathtt{out}}$ to be the masked output wires for the first evaluation.
5. Player P_1, P_2 broadcast $\mathsf{m}^1_{\mathtt{out}}$, if their broadcasts disagree then all players abort
6. Player P_3, P_4 broadcast $\lambda^1_{\mathtt{out}}$, if their broadcasts disagree then all players abort.
7. Players compute the output by using $\mathsf{m}^1_{\mathtt{out}}$ and $\lambda^1_{\mathtt{out}}$.

Figure 1: Main protocol in the hybrid model

This 4-party, randomized functionality is called by two distributors and two evaluators. No parties contribute any input. The functionality generates a vector of random masks as output for the distributors, and a secret-sharing of these masks for the evaluators.

Input: None.

Computation

1. Sample $\mathtt{seed}_1$ and $\mathtt{seed}_2$ uniformly at random. If the adversary corrupts D_2, allow him to specify the seeds.
2. For each wire $w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}$:
   (a) $\Lambda_{1,w} \leftarrow \mathsf{G}(\mathtt{seed}_1)$,
   (b) $\Lambda_{2,w} \leftarrow \mathsf{G}(\mathtt{seed}_2)$
   (c) $\lambda_w \leftarrow \Lambda_{1,w} + \Lambda_{2,w}$.
3. For each addition gate $(a, b, c, +)$: compute $\lambda_c \leftarrow \lambda_a + \lambda_b$.
4. For each multiplication gate $(a, b, c, \times)$
   (a) $\gamma_c \leftarrow \lambda_a \cdot \lambda_b$
   (b) $\Gamma_{1,c} \leftarrow \mathsf{G}(\mathtt{seed}_1)$
   (c) $\Gamma_{2,c} = \gamma_c + \Gamma_{1,c}$

Output

1. Output $\langle\gamma\rangle, \langle\lambda\rangle$ to E_1, E_2 (by sending $\mathtt{seed}_1$ to E_1 and $(\mathtt{seed}_2, \{\Gamma_{2,w}\}_{w\in\mathcal{W}_{\mathtt{mult}}})$ to E_2)
2. Output both $\mathtt{seed}_1$ and $\mathtt{seed}_2$ to both D_1, D_2

Malicious party: A malicious D_2 can choose the randomness.

Figure 2: $\mathcal{F}_{\mathsf{pre}}$: Ideal functionality for preprocessing

They both send Λ_1 and Γ_1 to E_1, and they both send Λ_2 and Γ_2 to E_2. E_1 and E_2 each verify the equality of the two values he received before proceeding to the evaluation phase. Note that after agreeing on the random string at the beginning of the procedure described above, D_1 and

Two distributors D_1, D_2 want to generate preprocessing for players E_1, E_2.

Creation

1. D_2 chooses two random seeds, $\mathtt{seed}_1$ and $\mathtt{seed}_2$, and sends them to D_1.
2. For each wire $w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}$:
   (a) $\Lambda_{1,w} \leftarrow \mathsf{G}(\mathtt{seed}_1)$,
   (b) $\Lambda_{2,w} \leftarrow \mathsf{G}(\mathtt{seed}_2)$
   (c) $\lambda_w \leftarrow \Lambda_{1,w} + \Lambda_{2,w}$.
3. For each addition gate $(a, b, c, +)$: compute $\lambda_c \leftarrow \lambda_a + \lambda_b$.
4. For each multiplication gate $(a, b, c, \times)$
   (a) $\gamma_c \leftarrow \lambda_a \cdot \lambda_b$
   (b) $\Gamma_{1,c} \leftarrow \mathsf{G}(\mathtt{seed}_1)$
   (c) $\Gamma_{2,c} = \gamma_c + \Gamma_{1,c}$

Distribution

1. D_1 sends $\mathtt{seed}_1$ to E_1 and $(\mathtt{seed}_2, \Gamma_2)$ to E_2.
2. D_2 sends $\mathtt{seed}_1$ to E_1 and $\mathsf{H}(\mathtt{seed}_2 || \Gamma_2)$ to E_2.
3. D_1 and D_2 output λ

E_1 **Reconstruction**

1. Receive $\mathtt{seed}_1$ from D_1 and D_2 and check they are the same. If not, abort.
2. $\{\Lambda_{1,w}\}_{w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}}, \{\Gamma_{1,w}\}_{w \in \mathcal{W}_{\mathtt{mult}}} \leftarrow \mathsf{G}(\mathtt{seed}_1)$
3. Output $(\{\Lambda_{1,w}\}_{w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}}, \{\Gamma_{1,w}\}_{w \in \mathcal{W}_{\mathtt{mult}}})$.

E_2 **Reconstruction**

1. Receive $(\mathtt{seed}_2, \Gamma_2)$ from D_1 and $\mathsf{H}(\mathtt{seed}_2 || \Gamma_2)$ from D_2 and check they are consistent. If not, abort.
2. $\{\Lambda_{2,w}\}_{w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}} \leftarrow \mathsf{G}(\mathtt{seed}_2)$
3. Output $(\{\Lambda_{2,w}\}_{w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}}, \{\Gamma_{2,w}\}_{w \in \mathcal{W}_{\mathtt{mult}}})$.

Notation

1. For each wire $w \in \mathcal{W}_{\mathtt{input}} \cup \mathcal{W}_{\mathtt{mult}}$: $\langle \lambda_w \rangle \leftarrow (\Lambda_{1,w}, \Lambda_{2,w})$
2. For each multiplication gate $(a, b, c, \times)$: $\langle \gamma_c \rangle \leftarrow (\Gamma_{1,c}, \Gamma_{2,c})$

Figure 3: Distributed Preprocessing of masked beaver triples

D_2 require no further communication with each other. Because one of the parties must be honest, the equality checks performed by E_1 and E_2 suffice to catch any malicious behavior. Note that this idea shares some similarity with the one by Mohassel et al. [MRZ15] in the three-party setting based on garbled circuit.

We do not present the pre-processing protocol in quite the way that was just described. Instead, an optimized variant with reduced communication complexity is presented in Fig. 3. First, instead of choosing and sending random strings of length $O(|C|)$, the two parties choose two short seeds for a PRG: we let $\Lambda_1 = \mathsf{G}(\mathtt{seed}_1)$, and $\Lambda_2 = \mathsf{G}(\mathtt{seed}_2)$. As before, $\lambda^1 = \Lambda_1 + \Lambda_2$. Since the value of γ^1 depends on λ^1, we cannot do the same thing there, but we can generate the shares Γ_1 from the same $\mathtt{seed}_1$, and then fix Γ_2 appropriately, using $O(|\mathcal{W}_{\mathtt{mult}}|)$ bits. This reduces the communication cost for each of the parties from $(2|\mathbb{F}|+1) \cdot |\mathcal{W}_{\mathtt{mult}}|$ to $2\kappa + |\mathbb{F}| \cdot |\mathcal{W}_{\mathtt{mult}}|$. Recall that D_1 and D_2 send identical copies of these values to an evaluator; we further reduce the communication by having one party send only a single hash of the pre-processing, which suffices for allowing each evaluator to verify the consistency of what he has received. Finally, note that this last optimization causes the communication costs to become unbalanced. Although we do not present it, note that we can re-balance the cost by having one party send the first half of Γ_2 together with a hash of the second half, while the other party sends the second half of Γ_2 together with a hash of the first half.

Evaluation. After receiving and verifying the consistency of the pre-processing, E_1 and E_2

There are two evaluators E_1, E_2 who want to evaluate a circuit C using preprocessing provided by distributors D_1, D_2. Each of the four players is assigned a set of input wires corresponding to his input to C.

Input

1. For each input wire w, one party holds input x_w.
2. For each input wire w, D_1 and D_2 hold λ_w.
3. For each $w \in \mathcal{W}$, E_1 and E_2 hold $\langle \lambda_w \rangle$.
4. For each multiplication gate $(a, b, c, \times)$, E_1 and E_2 hold $\langle \gamma_c \rangle \leftarrow \langle \lambda_a \cdot \lambda_b \rangle$.

Sharing Input Values

For each input wire w belonging to E_1 with value x_w:

1. D_1, D_2 both send λ_w to E_1. E_1 aborts if they are different.
2. E_1 sends $\lambda_w + x_w$ to E_2

For each input wire w belonging to D_1 with value x_w:

1. D_1 sends $\mathsf{m}_w \leftarrow x_w + \lambda_w$ to E_1 and E_2.
2. E_1 and E_2 verify that they each received the same value and abort if it is not the case.

The input of E_2 is processed similarly to the input of E_1.

The input of D_2 is processed similarly to the input of D_1.

Evaluation

For each gate (a, b, c, T) following topological order:

1. if $T = +$
   (a) $\mathsf{m}_c \leftarrow \mathsf{m}_a + \mathsf{m}_b$.
2. if $T = \times$
   (a) $\langle \mathsf{m}_c \rangle \leftarrow \mathsf{m}_a \cdot \mathsf{m}_b - \mathsf{m}_a \cdot \langle \lambda_b \rangle - \mathsf{m}_b \cdot \langle \lambda_a \rangle + \langle \lambda_c \rangle + \langle \lambda_a \cdot \lambda_b \rangle$
   (b) $\mathsf{m}_c \leftarrow \mathsf{open}(\langle \mathsf{m}_c \rangle)$

Figure 4: π_{eval} : Two-party Masked Evaluation

proceed to perform a mask-evaluation of the circuit, layer by layer. To begin, they first need masked input values for every input wire; these are of the form $\mathsf{m}_w \leftarrow \lambda_w + x_w$. For an input wire w held by $E \in \{E_1, E_2\}$, D_1 and D_2 send λ_w to E. E verifies that they each sent the same value: if not, he aborts. Otherwise, he computes $\lambda_w + x_w$ and sends it to the other evaluator. For input wire w belonging to $D \in \{D_1, D_2\}$, D sends $\lambda_w + x_w$ to E_1 and E_2. The evaluators compare values and abort if they don't agree.

For every gate $(a, b, c, +)$, E_1 and E_2 both locally compute $\mathsf{m}_c = \mathsf{m}_a + \mathsf{m}_b$. For every gate $(a, b, c, \times)$, they locally compute $\langle \mathsf{m}_c \rangle \leftarrow \mathsf{m}_a \cdot \mathsf{m}_b - \mathsf{m}_a \cdot \langle \lambda_b \rangle - \mathsf{m}_b \cdot \langle \lambda_a \rangle + \langle \lambda_c \rangle + \langle \lambda_a \cdot \lambda_b \rangle$. (Recall, they can compute the last term using $\langle \gamma_c \rangle$.) They then compute $\mathsf{m}_c \leftarrow \mathsf{open}(\langle \mathsf{m}_c \rangle)$ by exchanging their shares of m_c. At the conclusion of evaluation phase, one set of evaluators holds m^1, which is the set of masked values of all wires in the circuit, and the other group of parties hold m^2 after their evaluation phase.

Cross-checking. Note that during the evaluation phase, a malicious evaluator can modify the value on any $w \in \mathcal{W}_{\texttt{mult}}$ simply by changing his share of m_w before reconstructing the value. Therefore, before either group recovers output from their computation, they first compare their masking with the masking of the other evaluation. Of course, they cannot reveal the values on any wires while doing this check. Instead, for wire w that carries value x, each set of evaluators uses the masking from their evaluation, together with the masks that they generated for the other group during pre-processing, to compute

$$x + \lambda_w^1 + \lambda_w^2 = \mathsf{m}_w^1 + \lambda_w^2 = \mathsf{m}_w^2 + \lambda_w^1.$$

Input

1. P_1 has input $(\mathsf{m}_1^1, \lambda_1^2)$.
2. P_2 has input $(\mathsf{m}_2^1, \lambda_2^2)$.
3. P_3 has input $(\mathsf{m}_3^2, \lambda_3^1)$.
4. P_4 has input $(\mathsf{m}_4^2, \lambda_4^1)$.

Computation

1. P_1 computes $h_1 = H(\mathsf{m}_1^1 + \lambda_1^2)$,
   P_2 computes $h_2 = H(\mathsf{m}_2^1 + \lambda_2^2)$,
   P_3 computes $h_3 = H(\mathsf{m}_3^2 + \lambda_3^1)$,
   P_4 computes $h_4 = H(\mathsf{m}_4^2 + \lambda_4^1)$.
2. All players sends h_i to $\mathcal{F}_{\mathsf{4pc}}$, which outputs 1 if and only if $h_1 = h_3$ and $h_2 = h_4$.

Figure 5: π_{cross} : Cross Checking

They then compare these "doubly masked" values for consistency.

As in the case of pre-processing, we use a hash function where possible, in order to reduce the communication cost. Each party begins by computing a hash of the doubly masked wire values described above; for P_i, we denote this hash by h_i. The four parties then call an ideal functionality, $\mathcal{F}_{\mathsf{eq}}$, which takes input h_i, and outputs 1 if and only if $h_1 = h_3$, and $h_2 = h_4$.

Taking P_1 as example, he obtains m_1^1 during evaluation and λ_1^2 when acting as a D. He will then compute $h_1 = H(\mathsf{m}_1^1 + \lambda_1^2)$. For the other three parties, it is defined similarly as follows: superscripts denote the index of the masked evaluation and subscripts denote the identity of the party.

$$h_2 = H(\mathsf{m}_2^1 + \lambda_2^2), \qquad h_3 = H(\mathsf{m}_3^2 + \lambda_3^1), \qquad h_4 = H(\mathsf{m}_4^2 + \lambda_4^1)$$

To see why this suffices for providing security, suppose P_1 changes some masking during evaluation, effectively changing a wire value for him and P_2. In this case, the doubly masked evaluations of P_2 and P_4 are inconsistent, and $\mathcal{F}_{\mathsf{eq}}$ will return 0; intuitively, comparing these hash values is equivalent to checking the masked values wire by wire.

3.1 Concrete Performance

Here we briefly discuss the concrete performance of our protocol against the most related state-of-the-art protocol by Araki et al. [ABF+17]. As mentioned previously, our protocol requires 1.5 bits of communication per gate per party, a 4.5× improvement over their protocol. Let's see if the same applies to the computation cost. Note that in the protocol by Araki et al., the heaviest part of the computation is random shuffling, due to the use of the random bucketing technique in their paper. The rest are AES and hash computation, which can be hardware accelerated or very fast. Compared to their protocol, our protocol is much simpler and more efficient in terms of computation cost. The bulk of our computation is in the evaluation phase, where we do not need any random shuffling. For each 128 AND gates, each party only needs 6 calls to fix-key AES to implement the PRG, and roughly one call to a hash function. Araki et al. have a higher computational cost than we do, because of their random shuffle; since they are able to fill a 10Gbps LAN, our protocol will certainly have no problem filling the same pipe. We believe the computation cost will not be the bottleneck for any reasonable hardware configuration.

Our protocol for the cross checking appears in Fig. 5. It is in a hybrid world where the parties have access to a functionality, $\mathcal{F}_{\mathsf{eq}}$. We note that this functionality can be realized using any secure four party computation. The circuit needed to realize this functionality is small: it only

performs two equality computations on strings of length $O(\kappa)$. Nevertheless, in Section 5, we also demonstrate how we can bootstrap this functionality, communicating just a small constant number of bits, and using almost no computation.

3.2 Achieving one bit of communication using six parties

We note that if we use six players, we can maintain of the overhead of 6 bits communicated in total, thereby requiring each player to communicate just one bit per wire (on average). The idea is very simple. Two people agree on randomness for the preprocessing, and then each communicates the preprocessing material to two of the remaining four players. Those four parties now carry out two identical evaluations, in parallel, and cross check them with one another at the end. The communication overhead is still six bits per gate, but it is now divided among all six players.

4 Security Proof

4.1 Proof of Security for Preprocessing

Lemma 4.1. *The protocol in Fig. 3 for distributed pre-processing securely realizes the functionality of Fig. 2.*

Proof. Due to symmetry, we only prove the lemma for the following two cases: 1) D_1 is corrupt and 2) E_1 is corrupt.

Corrupted D_1. We will first describe our simulator $\mathcal{S}$.

1. $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{pre}}$ and obtains $\mathtt{seed}_1, \mathtt{seed}_2$. If the $\mathcal{A}$ chooses to input randomness, use $\mathcal{A}$'s choice.

2. $\mathcal{S}$ acts as honest D_2, E_1 and E_2 for the rest of the protocol using the seeds obtained above. If an honest E_1 or E_2 would abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{pre}}$.

Note that none of the parties in the protocol have input. Therefore the indistinguishability of the ideal-world protocol and the real-world protocol is immediate, given the observation that the protocol aborts in the real world protocol if and only if it aborts in the ideal world protocol.

Corrupted E_1. Note that E_1 performs only local computation after receiving messages from other parties. The simulator queries $\mathcal{F}_{\mathsf{pre}}$ and receives the seeds. He then simulates honest D_1 and D_2, sending $\mathtt{seed}_1$ on their behalf. If E_1 aborts, the simulator will send abort to $\mathcal{F}_{\mathsf{pre}}$ and aborts. Indistinguishability from the real-world protocol is immediate.

□

4.2 Proof of Security of the Main Protocol

Theorem 3. *Assuming H is a random oracle, our main protocol, in Fig. 1, securely realizes $\mathcal{F}_{\mathsf{4pc}}$ in the $(\mathcal{F}_{\mathsf{pre}}, \mathcal{F}_{\mathsf{eq}})$-hybrid model.*

Proof. In the following, we will prove the security of our main protocol assuming that P_1 is corrupted by $\mathcal{A}$. The simulator is as follows:

1. $\mathcal{S}$ honestly simulates the execution of $\mathcal{F}_{\mathsf{pre}}$. He sends P_1 his resulting output, and records the simulated mask values: λ^1, which will mask the wire values in the evaluation of P_3 and P_4, and λ^2, which will mask the wire values in the evaluation of P_1 and P_2.

2. $\mathcal{S}$ simulates the masking of input values 0 from P_2, P_3 and P_4 for use in P_1's evaluation with P_2, using mask values from λ^2. He receives three maskings of P_1's input: one for each of P_3 and P_4 for use in their evaluation, using mask values from λ^1, and one using values from λ^2, sent to P_2 for his own evaluation with P_1. If the values sent to P_3 and P_4 are not equal, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{4pc}}$ and terminates the simulation. Otherwise, $\mathcal{S}$ extracts the input sent to P_2, and the one sent to P_3 and P_4, using his knowledge of the masks; he notes if P_1 misbehaves by using inconsistent values in the two evaluations.

3. $\mathcal{S}$ acts honestly as P_2, P_3 and P_4 in both executions of the masked evaluations. $\mathcal{S}$ obtains m^1 by interacting honestly with P_1 on behalf of P_2 for the remainder of their evaluation. He obtains m^2 by simulating (internally) the remainder of the evaluation of P_3 and P_4. $\mathcal{S}$ notes if P_1 misbehaved in the masked evaluation produced by P_1 and P_2.

4. $\mathcal{S}$ collects P_1's input to $\mathcal{F}_{\mathsf{eq}}$. If P_1 sends the wrong input, if he misbehaved during input masking, or during his evaluation, $\mathcal{S}$ returns 0 from $\mathcal{F}_{\mathsf{eq}}$ to P_1, and sends abort to $\mathcal{F}_{\mathsf{4pc}}$. Otherwise, he returns 1 from $\mathcal{F}_{\mathsf{eq}}$ to P_1.

   Comment: In Section 5, we describe a more efficient, interactive protocol, π_{vcc}, which replaces the use of $\mathcal{F}_{\mathsf{eq}}$. To simulate our protocol when using π_{vcc}, we would proceed as follows, in place of the previous step. If $\mathcal{S}$ noted that that P_1 misbehaved during evaluation, or when sending his masked input, then $\mathcal{S}$ runs π_{vcc}, simulating the messages of P_2 and P_4 when using different (random) inputs from one another. Otherwise, he runs π_{vcc} as though P_2, P_3 and P_4 all use input $\mathsf{m}^1 \oplus \lambda^2$. If π_{cross} outputs 0, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{4pc}}$.

5. $\mathcal{S}$ uses the input extracted in Step 2 and sends it to $\mathcal{F}_{\mathsf{4pc}}$. He receives y and computes $\lambda^* = \mathsf{m}^1_{\mathsf{out}} + y$. $\mathcal{S}$ acts as P_3 and P_4 sending λ^* to $\mathcal{A}$.

Now we will show that the joint distribution of the output from $\mathcal{A}$ and honest parties in the ideal world are indistinguishable from these in the real world protocol.

1. Hybrid_1: Same as the hybrid protocol with $\mathcal{S}$ plays the role of honest players using their true input. (The resulting distribution is equivalent to that of the real world execution.)

2. Hybrid_2: Same as Hybrid_1, except that $\mathcal{S}$ obtains m^1 and λ^1 and compute $x_1 = \mathsf{m}^1_{\mathsf{in}} + \lambda^1_{\mathsf{in}}$. $\mathcal{S}$ sends x_1 to $\mathcal{F}_{\mathsf{4pc}}$, which returns y. In step 5, $\mathcal{S}$ acts as P_3 and P_4 and broadcasts $\lambda^* = \mathsf{m}^1_{\mathsf{out}} + y$.

3. Hybrid_3: Same as the Hybrid_2 except that all honest parties uses 0 instead of their true input.

□

5 Cross Check from Veto

In this section, we will demonstrate how to construct an efficient cross checking protocol based on a functionality for 4-party, logical OR, $\mathcal{F}_{\mathsf{or}}$. We sometimes call this a *veto functionality*, as the parties use the OR to "veto" the execution, by submitting a value of 1 (veto). The cross checking protocol

The protocol assumes access to an ideal functionality, $\mathcal{F}_{\mathsf{or}}$, for computing the logical OR of 4 input bits, each provided by one of the parties.

Input

1. P_1 has input $d_1 = \mathsf{m}_1^1 + \lambda_1^2$.
2. P_2 has input $d_2 = \mathsf{m}_2^1 + \lambda_2^2$.
3. P_3 has input $d_3 = \mathsf{m}_3^2 + \lambda_3^1$.
4. P_4 has input $d_4 = \mathsf{m}_4^2 + \lambda_4^1$.

Checking

1. P_1 samples a random seed and sends it to P_3.
2. P_1 (resp. P_3) send $H(d_1 \| \mathsf{seed})$ (resp. $H(d_3 \| \mathsf{seed})$) to P_2 and P_4
3. P_2 (resp. P_4) determines if it received the same value from P_1 and P_3. If it did, it will provide 0 to the $\mathcal{F}_{\mathsf{or}}$ functionality, and otherwise it will provide 1.
4. Repeat the previous instructions with the variable exchanged as follows: P_1 is switched with P_2, and P_3 is switched with P_4, d_1 is switched with d_3, and d_2 is switched with d_4.
5. Players call the $\mathcal{F}_{\mathsf{or}}$ functionality with the input that they were instructed to use in step 3.

Figure 6: Cross check protocol from veto

from Section 3 required a 4-party computation of $\mathcal{F}_{\mathsf{eq}}$, which compared 2 pairs of strings, each κ bits long. The improved cross checking protocol based on veto requires each party to compare two hashes locally, and then input a single bit to the veto functionality. While the cost of either of these protocols is small compared to the evaluation phase, the simplicity of the protocol here makes it hard to pass up. We also describe how to bootstrap $\mathcal{F}_{\mathsf{or}}$, using a variant of the protocol from Section 3, and requiring just 6 bytes of communication per party. Perhaps one of the nicest features of this bootstrapping, from a practical standpoint, is that it allows us to avoid any dependence on other MPC implementations.

Naive implementation of cross checking. A naive way of implementing cross checking is to have the two verifiers exchange their doubly masked evaluations, and compare them for inconsistencies. Unfortunately, this approach fails because the adversary can modify the values carried on any of the wires in his own evaluation, and determine precisely how the change impacted the evaluation of the circuit by subtracting his doubly masked evaluation from the other. The differences between these two doubly masked evaluations reveals the differences in the values carried on each wire in the two evaluations of the circuit.

Achieving secure and efficient cross checking. Our main observation for simplifying the cross check protocol is that, in the attack just described, P_1 will always cause the verification run by P_2 and P_4 to fail. This is because the evaluation of P_2 was also modified on wire w, but he will not modify λ_w^1 the way P_1 did. If the output of the equality test between P_1 and P_3 were hidden from P_1, shown only to P_2 and P_4, and, symmetrically, if P_1 only saw the result of their verification (which he already knows), then we can remove the bit of leakage. Specifically, each P_i learns a single bit, b_i, indicating whether the *other* verifying set passed the equality test. The four parties then run a secure protocol that computes the logical OR of these 4 bits. They can do this using any existing 4-party protocol.

One verification group reveals the equality of their masked evaluations to the other verification group as follows. (1) They agree on a random seed, (2) they hash it together with their doubly masked evaluation, and (3) they send the hash output to the players of the other verification group. The players in the other verification group can compute equality by simply checking that the hashes

they receive are the same.

Note that $\mathcal{F}_{\mathsf{or}}$ is a constant size circuit, and it likely does not matter which four party secure computation we use to realize it. Still, it is interesting to note that we can actually bootstrap this computation with another variant of our own protocol. In the protocol just previously described, letting $d_{i,w}$ denote the doubly masked value held by P_i for wire w, the parties effectively compute $\bigvee_{w \in \mathcal{W}}(d_{1,w} \neq d_{3,w}) \vee \bigvee_{w \in \mathcal{W}}(d_{2,w} \neq d_{4,w})$, where the hash value received by P_1 and P_3 (resp. P_2 and P_4) reveals the first (resp. second) disjunction of size $|\mathcal{W}|$ to P_1 and P_3 (resp. P_2 and P_4). The disjunction in the middle is where we use $\mathcal{F}_{\mathsf{or}}$. Following the same discussion above, the reader can verify that it is also secure to compute $\bigvee_{w \in \mathcal{W}}((d_{1,w} \neq d_{3,w}) \vee (d_{2,w} \neq d_{4,w}))$. This can be achieved by having the four parties check the equality of gates in topological order by immediately exchanging the results of every equality check, rather than "batching them" with a hash function at the end of the evaluation. Removing the hash function in this way increases the communication to $O(|C|)$, so we would not want prefer to use this as our cross-checking protocol. But to bootstrap $\mathcal{F}_{\mathsf{or}}$, which has just 3 gates, there is no need for the use of a hash function.

Security of Veto Cross Check

(Sketch.) Our main protocol is secure if we replace the cross checking in the main protocol with the cross checking described in this section.

If the adversary acted maliciously during the masked evaluation, then it is clear that the verification group that does not contain the corrupt player (i.e. the honest verification group) will have inconsistent evaluations. As a result, the simulator can run the cross checking on behalf of the honest players as though the player in the honest verification group had inconsistent evaluations. In this case, the honest player in the same validation group as the corrupt player will always provide a veto. As a result, the simulator can safely always provide a simulated output of veto from $\mathcal{F}_{\mathsf{or}}$, sends abort to $\mathcal{F}_{\mathsf{4pc}}$, and the result is indistinguishable from a real execution.

If instead the corrupt player only misbehaves in the cross checking, the only possible deviation is to send the wrong hash value. In this case, the simulator can compute whether the corrupt player misbehaved by analyzing the hash value that he sent, together with the seed. The simulator knows that both players in the honest verification group will veto. As a result, the simulator can simply provide a simulated output of veto from $\mathcal{F}_{\mathsf{or}}$, submit abort to $\mathcal{F}_{\mathsf{4pc}}$, and the result is indistinguishable from a real execution.

Finally, if the adversary never deviates from the protocol, the simulator accepts the adversary's input to $\mathcal{F}_{\mathsf{or}}$ and sends it back to him as the output of $\mathcal{F}_{\mathsf{or}}$. If this value is a veto, the simulator sends abort to $\mathcal{F}_{\mathsf{4pc}}$, and otherwise, he submits the adversary's input to $\mathcal{F}_{\mathsf{4pc}}$, and simulates the opening of the output just as in Section 4.

The view in the real and ideal world are indistinguishable since (1) the simulator can always determine if there is a veto or not for the functionality based on the behavior of the adversary and (2) the random oracles hides inputs from the other verification group.

6 Adding Robustness

We can make our protocol robust against a single cheater. We note that it is quite simple to strengthen our original protocol so that it is *fair*. If the malicious party aborts before anyone sends the output wire masks, then nothing is learned, and all parties can safely abort. If the adversary

Two distributors D_1, D_2 want to generate preprocessing for players E_1, E_2. We assume D_1 holds a key pair for a public key, committing encryption scheme, and that both hold key pairs for a digital signature scheme. The 3 public keys are known by all parties. We let $(\mathsf{pk}, \mathsf{sk})$ denote the encryption/decryption keys of D_1, and $(\mathsf{vk}_i, \mathsf{sk}_i)$ denote the verifying/signing keys of D_i.

Protocol

1. D_2 chooses $(\mathsf{seed}_1, \mathsf{seed}_2, \mathsf{r}_{\mathsf{com}})$ at random and broadcasts $\mathsf{enc}(\mathsf{pk}, \mathsf{seed}_1 || \mathsf{seed}_2 || \mathsf{r}_{\mathsf{com}})$. If he fails to do so, or if he broadcasts an invalid ciphertext, the other 3 parties run a 3-party, semi-honest protocol, where D_1 generates the preprocessing.
2. D_1 recovers $(\mathsf{seed}_1, \mathsf{seed}_2, \mathsf{r}_{\mathsf{com}})$.
3. Each D_i computes the preprocessing that was described in Fig. 3.
4. Each D_i computes $\mathsf{commit}(\{\lambda_w\}_{w \in \mathcal{W}_{\text{output}}}; \mathsf{r}_{\mathsf{com}})$. He includes these commitments in the preprocessing material.
5. Each D_i signs the preprocessing material: $\sigma_{1,i} = \mathsf{sign}(\mathsf{sk}_i; \mathsf{seed}_1)$ and $\sigma_{2,i} = \mathsf{sign}(\mathsf{sk}_i; \mathsf{seed}_2 || \Gamma_2)$. He sends $(\mathsf{seed}_1, \sigma_{1,i})$ to E_1, and $(\mathsf{seed}_2, \Gamma_2, \sigma_{2,i})$ to E_2.
6. E_j receives $(m_{j,1}, \sigma_{j,1})$ and $(m_{j,2}, \sigma_{j,2})$. He checks whether $\mathsf{vrfy}(\mathsf{pk}_1, \sigma_{j,1}) = \mathsf{vrfy}(\mathsf{pk}_2, \sigma_{j,2}) = 1$, and whether $m_{j,1} = m_{j,2}$.
   - If one of the signatures does not verify, E_j continues the protocol using only the preprocessing material that was validly signed.
   - If both signatures verify, but $m_{j,1} \neq m_{j,2}$,
     - E_j broadcasts the two signed messages.
     - D_2 broadcasts $(\mathsf{seed}_1, \mathsf{seed}_2)$, together with the encryption randomness used in the Step 1. All honest parties can now determine whether D_1 or D_2 misbehaved. They eliminate the guilty party and execute a 3-party, semi-honest protocol.
   - E_j outputs $m_{j,1}$.

Figure 7: Robust Preprocessing

aborts after learning the output masks, his partner can still reveal the output for the other two evaluators. The only necessary modification is to prevent the malicious distributor from changing his output masks, revealing output values that conflict with what his partner reveals. This is easily handled by having all parties commit to their output masks prior to the evaluation: if the two distributors use the same randomness in their commitments, the evaluators can verify that they have both committed to the same mask value.

The main challenge in achieving robustness is that we cannot simply abort when we detect improper behavior, even if the output has not been revealed yet. Instead, we have to ensure that all honest parties correctly identify a misbehaving party, or at least a pair of parties that contains the adversary. To facilitate this, we make several adjustments. First, we modify the preprocessing protocol so that it either allows everyone to identify the adversary, or it ensures that both evaluators receive good preprocessing material. The robust preprocessing appears in Fig. 7. We then modify the input sharing to make it robust; the input sharing in Section 3 would trigger an abort if any party used different inputs in the two executions, but it would not allow the others to determine who cheated. After receiving the preprocessing material and the masked inputs, the evaluators continue the evaluation protocol from Section 3 until each party has a masking of the circuit. They then perform a robust variant of the cross checking protocol. In this variant, the parties cross check gate by gate, and if they ever find an inconsistency, they run a sub-routine to identify a pair of

Input phase

1. For each input wire w
   (a) Suppose $P_1 = E_1$ is the player who provides input x_w for wire w (we can generalize this to the other parties)
   (b) E_1 awaits the mask λ_w^2 from D_1, D_2, as well as signatures on λ_w^2.
      i. If E_1 receives a value for λ_w^2 without a signature from D_i then he ignores the mask that D_i sent him.
      ii. Otherwise, he received inconsistent masks, E_1 broadcasts the signed masks (thus identifying which evaluation group contains a cheater). If the players receive two different masks with valid signatures, they run the protocol, using only the masked evaluation of E_1, E_2.
   (c) E_1 broadcasts $\mathsf{m}_w = x_w + \lambda_w^1 + \lambda_w^2$.
   (d) E_1, E_2 set $\mathsf{m}_w^1 = \mathsf{m}_w - \lambda_w^1$ while D_1, D_2 set $\mathsf{m}_w^2 = \mathsf{m}_w - \lambda_w^2$.

Evaluation

Each evaluation group, using their own masked evaluation, as well as the share of the masks they received from distributors do the following:

For each gate (a, b, c, T) following topological order:

(a) if $T = +$
   i. $\mathsf{m}_c \leftarrow \mathsf{m}_a + \mathsf{m}_b$.

(b) if $T = \times$
   i. $\langle \mathsf{m}_c \rangle \leftarrow \mathsf{m}_a \cdot \mathsf{m}_b - \mathsf{m}_a \cdot \langle \lambda_b \rangle - \mathsf{m}_b \cdot \langle \lambda_a \rangle + \langle \lambda_c \rangle + \langle \lambda_a \cdot \lambda_b \rangle$
   ii. $\mathsf{m}_c \leftarrow \mathsf{open}(\langle \mathsf{m}_c \rangle)$

Cross Check

For every wire $w \in \mathcal{W}_{\mathtt{mult}} \cup \mathcal{W}_{\mathtt{input}}$, ordered by depth in the circuit.

For each verification group $V \in \mathcal{V}$

1. $(V_1, V_2) \leftarrow V$
2. V_1 send $d \leftarrow \mathsf{m}_w^1 + \lambda_w^1$ to V_2.
3. V_2 broadcasts (error) if $d \neq \mathsf{m}_w^2 + \lambda_w^2$.
4. If a player in V broadcasts (error), run complaint(w).
   (a) If the complaint phase returns ($\mathsf{corrupt}, P_1, P_2$), P_1, P_2 broadcast decommitments to λ_w^1 for each output wire w. P_3, P_4 compute $\mathsf{m}_w^2 - \lambda_w^1$, broadcast the result, and the protocol terminates.
   (b) If the complaint phase returns ($\mathsf{corrupt}$, P_3, P_4), P_3, P_4 broadcast decommitments to λ_w^2 for each output wire w. P_1, P_2 compute $\mathsf{m}_w^1 - \lambda_w^2$, broadcast the result, and the protocol terminates.
   (c) If the complaint phase returns ($\mathsf{corrupt}$, verifier), set $\mathcal{V} \leftarrow \mathcal{V} \setminus V$ and restart the protocol with the updated $\mathcal{V}$.

Output

For each output wire w,

1. Players P_1, P_2 broadcast the decommitment to λ_w^1.
2. Players P_3, P_4 broadcast the decommitment to λ_w^2.
3. P_1 and P_2 broadcast $\mathsf{m}_w^1 - \lambda_w^2$. Denote these values by ($\mathsf{out}_1, \mathsf{out}_2$).
4. P_3 and P_4 broadcast $\mathsf{m}_w^2 - \lambda_w^1$. Denote these values by ($\mathsf{out}_3, \mathsf{out}_4$).
5. All parties output $\mathsf{Majority}(\mathsf{out}_1, \mathsf{out}_2, \mathsf{out}_3, \mathsf{out}_4)$.

Figure 8: Robust Evaluation

parties that contains the adversary.[1] Input sharing, evaluation, and robust cross checking are fully described in Fig. 8. We give a detailed overview of these changes below.

Robust preprocessing: To make the preprocessing robust, one of the two distributors, D_2, starts by committing to the randomness that will be used in the preprocessing. This commitment is constructed by broadcasting a committing encryption under the public key of D_1. The randomness used in the preprocessing is denoted by $(\mathtt{seed}_1, \mathtt{seed}_2, \mathsf{r}_{\mathsf{com}})$: $\mathtt{seed}_1$ and $\mathtt{seed}_2$ are used to create masks, just as in Section 3. $\mathsf{r}_{\mathsf{com}}$ is used to construct a commitment to the output masks, which is then included in the preprocessing output.

After generating the preprocessing material, D_1 and D_2 each sign a copy of the output before sending it to E_1 and E_2. If they send conflicting values to E_1, the signatures allow E_1 to convince the other honest parties that one of D_1 or D_2 is malicious. The honest one of the two can now be exonerated: D_2 broadcasts the randomness used to encrypt the preprocessing randomness. E_1 broadcasts their view, and the honest parties can check the validity of of the messages sent by D_1 and D_2. After removing the malicious party, the remaining three parties can run a semi-honest protocol in which one party supplies the preprocessing, the other two perform the evaluation, and no checking needs to be performed.

One other case of note deserves mention: suppose E_1 receives nothing[2] from, say, D_1. In this case, because there is no signature, E_1 cannot prove that D_1 or D_2 is malicious: it is equally possible that E_1 is himself malicious, and that he made the problem up. In this case, though, E_1 does need to persuade anybody. Because E_1 knows that D_1 is malicious, E_1 can simply continue the protocol using the preprocessing he received from D_2.

Robust input sharing: Let P_1 and P_2 perform distribution for P_3, P_4, and vice versa. Recall that Section 3, P_1 shares input x_w on wire w with P_3, P_4 by using the mask λ_w^1 that he and P_2 generated together. He shares his input with P_1, for their own evaluation, by using λ_w^2, which he receives from P_3, P_4. As written, nothing prevents him from sharing inconsistent values among the parties, and nothing prevents those parties from pretending he did so. To fix this, we first require P_3, P_4 to each sign λ_w^2, which allows P_1 to broadcast a proof of inconsistency when necessary. Then, P_1 signs and broadcasts his doubly masked input: $\mathsf{m}_w = x_w + \lambda_w^1 + \lambda_w^2$. P_2 computes $\mathsf{m}_w - \lambda_w^1$ for use in his evaluation with P_2. P_3, P_4 each compute $\mathsf{m}_w - \mathsf{m}_w^2$ for use in their evaluation.

Robust cross checking: Instead of cross checking the hashes of the full circuit maskings, the parties instead cross check gate by gate, starting at the input layer, and proceeding topologically through the circuit. This protocol begins with a pass over the circuit, one layer at a time, with the parties comparing their doubly masked values to locate the first gate at which the two evaluations depart from one another. Consider the case where P_3 decides that the two masked evaluations of some gate are inconsistent, and initiates a complaint. This can be due to one of the following cases:

1. The masked evaluation performed by P_1 and P_2 is invalid.
2. The masked evaluation performed by P_3 and P_4 is invalid.
3. Both evaluations were executed correctly, but either P_1 modified his input to cross-checking (i.e. his reported masked evaluation), or P_3 complained for no valid reason.

[1] To reduce communication of the robust cross checking, we can iteratively apply our cross check protocol from Section 5, performing a binary search on the masked circuit layers until we find the problematic layer. We then repeat that, performing a binary search within the problematic layer to find the problematic gate. This would yield a worst-case communication cost of $O(\kappa \log |C|)$. For simplicity, we describe the protocol as operating gate per gate.

[2] equivalently, something that is not validly signed

Complaint Subprotocol

The *complaint* subprotocol is initiated for output wire w for a multiplication gate g_w when a player C has complained that the two masked evaluations were inconsistent on that wire. We denote by $V \in \mathcal{V}$ the verification group that contains C. This subprotocol allows the parties to identify either an evaluation group or a verification group that contains a cheater.

Complaint

1. $\mathcal{E}_1 \leftarrow \{P_1, P_2\}$, $\mathcal{E}_2 \leftarrow \{P_3, P_4\}$, $\mathcal{D}_1 \leftarrow \{P_3, P_4\}$, $\mathcal{D}_2 \leftarrow \{P_1, P_2\}$
2. Players run the validation functionality for wire w using $\mathcal{E}_1$ as the evaluators and $\mathcal{D}_1$ as the distributors.
3. Players run the validation functionality for wire w using $\mathcal{E}_2$ as the evaluators and $\mathcal{D}_2$ as the distributors.
4. If any of the calls to the validation functionality results in the functionality returning (corrupt,P_i,P_j), return the same. Otherwise, return (corrupt, verifier).

Validation

E_1, E_2, D_1 and D_2 want to verify that the masked evaluation of E_1, E_2 was done correctly for the gate g_w with output wire w.

Input

1. E_1, E_2 each input their masked evaluation for the 3 wires of g_w: $(\mathsf{m}_a, \mathsf{m}_b, \mathsf{m}_c)$.
2. D_1, D_2 each input the masks that they generated g_w: $(\lambda_a, \lambda_b, \lambda_c)$.

Functionality

1. If E_1 and E_2 provided distinct inputs, return (corrupt,E_1,E_2) and halt.
2. If D_1 and D_2 provided distinct inputs, return (corrupt,D_1,D_2) and halt.
3. If $(\mathsf{m}_a - \lambda_a) \cdot (\mathsf{m}_b - \lambda_b) + \lambda_c = \mathsf{m}_c$, then output (valid). Otherwise output (corrupt,E_1,E_2).

Figure 9: Complaint

If the honest players know that the first case holds, then the corrupt player is either P_1 or P_2. They can therefore use the evaluation of P_3 and P_4 to determine their output. By the same argument, if the players know that the second case holds, they can all safely use the evaluation of P_1 and P_2 to produce the output. Finally, if the players know they are in the third case, they know that the malicious party is either P_1 or P_3. In this case, they do not dismiss either evaluation, but they continue the cross checking using only between P_2 and P_4; since P_2 and P_4 are honest, their cross-checking suffices for ensuring a valid computation.

When someone detects an inconsistency in the cross checking of a gate, the parties execute a *complaint* subprotocol (See Fig. 9) to determine which of the above cases hold. In this subprotocol, the parties use an ideal functionality, which can later be bootstrapped generically using any MPC with identifiable abort. We stress that the circuit implementing this functionality is small: it only needs to be executed on a single gate, and it used at most twice in a computation. The functionality is called once for each of the two evaluations. In each instance, the two evaluators provide their masked input and masked output for the gate, while the two distributors provide the masks that they created for the gate. If the evaluators do not provide the same masked values then the functionality indicates that the evaluation set contains the cheater. If the distributors do not provide the same masks then the functionality indicates that the cheater is in the distribution set. Otherwise, the functionality uses the masked wire values and the mask values to check whether the gate evaluation was performed correctly. If the masked evaluation was invalid, the ideal functionality indicates that the evaluation set contains the cheater. Finally, if no error is detected, then the

functionality indicates this, and the parties conclude that either the party that raised the alarm is malicious, or his partner in the cross-checking is malicious (case 3 above).

6.1 Robust Evaluation Simulator

Theorem 4. *If the robust evaluation protocol is instantiated using a CCA-Secure public-key committing encryption scheme then it securely realizes $\mathcal{F}_{\mathsf{4pc}}$ in the random oracle model. In addition, the protocol is robust.*

Simulator for robust preprocessing when D_2 is corrupt.

1. Await that D_2 broadcast the ciphertext c. If the ciphertext is invalid then the simulator submits a default input value to $\mathcal{F}_{\mathsf{4pc}}$ on behalf of the adversary, and terminates. (This corresponds to the honest parties removing the adversary from the computation, upon agreeing that he is malicious.)
2. Recover $(\mathsf{seed}_1 \| \mathsf{seed}_2 \| \mathsf{r}_{\mathsf{com}}) \leftarrow Dec(sk, c)$.
3. The simulator computes the preprocessing and broadcasts $\mathsf{commit}(\{\lambda_w\}_{w \in \mathcal{W}_{\text{output}}}; \mathsf{r}_{\mathsf{com}})$ on behalf of D_1.
4. Simulator awaits that D_2 sends the preprocessing material and signatures on the preprocessing material to each player. Then,
   (a) For each evaluator, if D_2 sent an invalid signature to the given evaluator, the simulator ignores what D_2 sent.
   (b) Otherwise, if D_2 sent invalid preprocessing to either evaluator, then simulate the broadcast from the given evaluator of the the signed preprocessing and determining that D_2 misbehaved. The simulator notes that D_2 was identified as a cheater.

Simulator for robust preprocessing when D_1 is corrupt. Same as the simulation for D_2 except that the simulator broadcasts the encryption of the randomness to D_1.

Simulator for robust preprocessing when an evaluator is corrupt. The simulator chooses randomness and simulates the three honest players. If an evaluator sends a message claiming he received inconsistent preprocessing, but the signed messages he forwards do not substantiate his claim, the simulator sends a default input to $\mathcal{F}_{\mathsf{4pc}}$ and terminates. (Technically, we did not describe in our protocol that the other parties remove the evaluator when he does this, because we felt it would unnecessarily complicate the protocol description.)

Indistinguishability of robust preprocessing.

1. In the case where the distributor is corrupt, we claim the view in the real and ideal worlds are indistinguishable. If the distributor deviates from the protocol, it is either ignored (if it does not send a signature with the preprocessing it shares), or it is eliminated from the computation (if it sends bad preprocessing with a valid signature). The committing property of the encryption scheme guarantees that he gets caught if he signs and sends a wrong value.

2. The only message sent by an evaluator is (possibly) to complain about inconsistent preprocessing. If the evaluator is corrupt, then in both the ideal and real world, the complaint would be ignored (due to the unforgeability of the underlying signature scheme).

Simulator for input sharing.

1. If the corrupt player is providing input as an evaluator,
   (a) The simulator provides the signed masks from the other distributors.
   (b) The simulator awaits that the corrupt player broadcasts a double masking m_w. The simulator then computes the input of the corrupt player from the masks that were produced in the preprocessing and the double masking that the corrupt player sent.
2. If the corrupt player is a distributor, and the input wire belongs to an evaluator,
   (a) The simulator awaits that the corrupt player sends out a mask to the evaluator. If the mask is signed with the corrupt player's signature, and is not the value produced in the preprocessing, then the simulator produces a broadcast of the conflicting, signed masks. The simulator provides the default value to $\mathcal{F}_{\mathsf{4pc}}$ on behalf of the corrupt player and terminates.
3. If the corrupt player is an evaluator, and the input wire belongs to the other evaluator, the simulator broadcasts the doubly masked input.

Indistinguishability of input phase. We argue that since the view until the end of the preprocessing phase in the ideal world is indistinguishable from the view until the end of the preprocessing phase in the real world, then the views are also indistinguishable up through the end of the input phase. In the real and ideal world, when the distributor is corrupt, any deviation would either be ignored, or would result in the dealer being caught and eliminated from the computation. If the evaluator is corrupt, and he broadcasts an invalid complaint, he is eliminated due to the unforgeability of the underlying signature scheme.

Simulator for evaluation. The simulator of the evaluation step follows the same steps as the simulator for the masked evaluation in the main protocol. In particular, the simulator stores if the corrupt player misbehaved during his evaluation. We argue that since the view until the end of the input phase in the ideal world is indistinguishable from the view until the end of the input phase in the real world, then the views are also indistinguishable up through the end of the evaluation phase.. This holds from the fact that our main protocol (in particular the masked evaluation part) is secure.

Simulator for cross check. For every multiplication wire $w \in \mathcal{W}_{\texttt{mult}}$,

1. if the corrupt evaluator had previously sent a wrong value in the evaluation of wire w,
   (a) The simulator broadcasts (error) on behalf of the verifiers that are not in the same verification group as the corrupt player. (He might also do with the player that is in the same verification group as him.)

(b) The simulator receives $(\lambda_a, \lambda_b, \lambda_c)$ and $(\mathsf{m}_a, \mathsf{m}_b, \mathsf{m}_c)$ from the adversary, intended for the first and second calls to the validation functionality, respectively (and without loss of generality). If $(\lambda_a, \lambda_b, \lambda_c)$ are inconsistent with the values simulated during preprocessing, the simulator implicates the adversary (and his partner) when simulating the output of the first call to the validation functionality. In either case he implicates the adversary (and his partner) in the simulated output of the second call to the validation functionality.

The simulator then runs the protocol on behalf of the honest players using the honest evaluation group's masked evaluation.

2. Otherwise:

   (a) if simulating V_1, the simulator checks to see if the adversary sends a wrong doubly masked value to his partner: $\mathsf{m}^1_w + \lambda^1_w$.

   (b) if simulating V_2, the simulator checks to see if the adversary broadcasts (error).

   The simulator receives $(\lambda_a, \lambda_b, \lambda_c)$ and $(\mathsf{m}_a, \mathsf{m}_b, \mathsf{m}_c)$ from the adversary, intended for the first and second calls to the validation functionality, respectively (and without loss of generality). If $(\lambda_a, \lambda_b, \lambda_c)$ are inconsistent with the values simulated during preprocessing, the simulator implicates the adversary (and his partner) when simulating the output of the first call to the validation functionality. If $(\mathsf{m}_a, \mathsf{m}_b, \mathsf{m}_c)$ are inconsistent with simulated masked values of the evaluation phase, the simulator implicates the adversary (and his partner) when simulating the output of the second call to the validation functionality. If he is not implicated in either instance, then any future messages he might send during cross checking are ignored.

Indistinguishability of cross check. We argue that since the view until the end of the evaluation phase in the ideal world is indistinguishable from the view until the end of the evaluation phase in the real world, then the views are also indistinguishable up through the end of the cross check. If the corrupt player's evaluation group is deemed corrupt, then the protocol in the real world would dictate that the corrupt player no longer receive messages during the cross check phase. Therefore, it is clear that after the elimination has taken place, the views in the real and ideal world are indistinguishable.

We now claim that the validation function eliminates the adversary's evaluation set in the real world, if and only if the simulator implicates the adversary's evaluation set in the ideal world. Note that the simulator can detect if the adversary has modified any wire in the evaluation, as well as whether his input to the validation function is inconsistent with his partner's input. The reader can verify by inspection that the claim holds. Since the complaint phase consists of just two calls to the validation functionality, it follows that the adversary's view in the complaint phase is identically distributed in the two worlds. By the previous note, after this point, the cross check in the real and ideal worlds would be indistinguishable.

To complete the argument that the adversary's view is correctly simulated through the end of the cross check phase, we argue that, prior to being eliminated, the simulated view in the cross check phase is sampled from the same distribution as his view in the real world. This follows because he only sees doubly masked wire values, which are computationally indistinguishable from uniformly distributed strings (because they are generated using a PRG).

Simulator for output phase. The output phase is the easiest to simulate.

1. First the simulator queries the ideal functionality with the adversary's input and receives an output.
2. The simulator selects masks for the honest evaluation group so that the sum of the output and the masks of the honest evaluation group is equal to the masked evaluation of the corrupt player. The simulator then "broadcasts" decommitments to the masks of the honest evaluation group.
3. The simulator selects masked evaluation for the honest evaluation group so that the sum of the output and the masks of the corrupt player add up to the masked evaluation. The simulator then "broadcasts" the masked evaluations.

Indistinguishability of output phase. We now argue that the output distribution, conditioned on the adversary's view, is indistinguishable in the two worlds. We have already argued that the adversary is caught if he ever manipulates his evaluation. The reader can verify that whenever a transcript results in the use of a default adversarial input in the real world, the simulator submits default input in the ideal world. If the adversary never changes the masked values, then the input used in both worlds is the one he committed to in the input sharing phase.

References

[ABF+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[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, *ACM CCS 08*, 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, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

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

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[CKMZ14] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 513–530, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 378–394, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

[DIK+08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[DNNR17] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. LNCS, pages 167–187, Santa Barbara, CA, USA, 2017. Springer, Heidelberg, Germany.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255, Paris, France, May 8–12, 2017. Springer, Heidelberg, Germany.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710, Victoria, British Columbia, Canada, May 4–6, 1992. ACM Press.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

[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 ACM STOC*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[HKE12] Yan Huang, Jonathan Katz, and David Evans. Quid-Pro-Quo-tocols: Strengthening semi-honest protocols with dual execution. In *2012 IEEE Symposium on Security and Privacy*, pages 272–284, San Francisco, CA, USA, May 21–23, 2012. IEEE Computer Society Press.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS 17*, pages 259–276. ACM Press, 2017.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[MF06] Payman Mohassel and Matthew Franklin. Efficient polynomial operations in the shared-coefficients setting. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 44–57, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 591–602, Denver, CO, USA, October 12–16, 2015. ACM Press.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*,

pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[RBO89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85, Seattle, WA, USA, May 15–17, 1989. ACM Press.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS 17*, pages 21–37. ACM Press, 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *ACM CCS 17*, pages 39–56. ACM Press, 2017.