Linear Overhead Robust MPC with Honest Majority Using Preprocessing

Ashish Choudhury\textsuperscript{1}, Emmanuela Orsini\textsuperscript{2}, Arpita Patra\textsuperscript{3}, and Nigel P. Smart\textsuperscript{2}

\textsuperscript{1} International Institute of Information Technology, Bangalore, India,
\textsuperscript{2} Dept. Computer Science, University of Bristol, Bristol, United Kingdom,
\textsuperscript{3} Dept. of Computer Science and Automation, Indian Institute of Science, Bangalore, India.

Abstract. We present a technique to achieve $\mathcal{O}(n)$ communication complexity per multiplication for a wide class of robust practical MPC protocols. Previously such a communication complexity was only known in the case of non-robust protocols in the full threshold, dishonest majority setting. In particular our technique applies to robust threshold computationally secure protocols in the case of $t < n/2$ in the pre-processing model. Surprisingly our protocol for robust share reconstruction with $\mathcal{O}(n)$ communication applies for both synchronous and asynchronous communication models. We go on to discuss implications for asynchronous variants of our resulting MPC protocol.
1 Introduction

Determining the communication complexity, in terms of the number of parties, of protocols for Multi-Party Computation (MPC) is a task which is both interesting from a theoretical and a practical standpoint. It is a folklore belief that the complexity should be essentially $O(n)$ per multiplication, where $n$ is the number of parties. However, “most” robust secret-sharing based protocols which are practical has complexity $O(n^2)$.

To understand the problem notice that apart from the protocols for entering parties inputs and determining parties outputs, the main communication task in secret-sharing based MPC protocols is the evaluation of the multiplication gates (we assume a standard arithmetic circuit representation of the function to be computed for purely expository reasons, in practice other representations may be better). If we consider the classic information-theoretically secure semi-honest sub-protocol for multiplication gates when $t < n/2$ (locally multiply the shares, recombine and then recombine) we require $O(n^2)$ messages per multiplication gate [6][19]. This is because each party needs to send the shares representing their local multiplication to every other party, thus requiring $O(n^2)$ messages, and hence $O(n^2)$ bits if we only look at complexity depending on $n$.

Even if we look at such protocols in the pre-processing model, where so-called “Beaver multiplication triples” are produced in an offline phase [3], and we are primarily concerned about the communication complexity of the online phase, a similar situation occurs. In such protocols, see for example [14], the standard multiplication sub-protocol is for each party to broadcast a masking of their shares of the gate input values to every other party. This again has $O(n^2)$ communication complexity.

In the SPDZ protocol [16], for the case of non-robust maliciously secure MPC (with abort) in the dishonest majority setting, an online communication complexity of $O(n)$ was achieved. This is attained by replacing the broadcast communication of the previous method with the following trick. For each multiplication gate one party is designated as the “reconstructor”. The broadcast round is then replaced by each party sending their masked values to the reconstructor, who then reconstructs the value and then sends it to each party. This requires exactly $2 \cdot n$ messages being sent, and is hence $O(n)$. However, this protocol is only relevant in the dishonest majority setting as any dishonest behaviour of any party is subsequently detected via the SPDZ MAC-checking procedure in which case the protocol aborts. If we require a robust protocol we appear unable to adopt this procedure.

With $t < n/3$, information-theoretically secure online protocols with $O(n)$ communication per multiplication are presented in [4][15]. The basic idea there is a new method of reconstructing a batch of $\Theta(n)$ secret-shared values with $O(n^2)$ communication complexity, thus providing a linear overhead. However, the method is tailor-made only for $t < n/3$ (as it is based on the error-correcting capability of the Reed-Solomon (RS) codes) and will not work with $t < n/2$. Hence with $t < n/2$ in the computational setting, a new technique to obtain $O(n)$ online complexity is needed. It is to this question that we address in this paper.

Before proceeding we pause to examine the communication complexity of the offline phase of such protocols. It is obvious that in the case of a computationally secure offline phase one can easily adapt the somewhat homomorphic encryption (SHE) based offline phase of SPDZ to the case of Shamir secret sharing when $t < n/2$. In addition one can adapt it to generate SPDZ style MACs or BDOZ [7] (a.k.a. pairwise MACs). In [16] the offline communication complexity is given as $O(n^2/s)$ in terms of the number of messages sent, where $s$ is the “packing” parameter of the SHE scheme. As shown in the full version of [17], assuming a cyclotomic polynomial is selected which splits completely modulo the plaintext modulus $p$, the packing parameter grows very slowly in terms of the number of parties (for all practical purposes it does not increase at all). In addition since $s$ is in the many thousands, for all practical purposes the communication complexity of the offline phase is $O(n)$ in terms of the number of messages. However, each message is $O(s)$ and so the communication complexity in terms of the number of bits is still $O(n^2)$.

In [20], a computationally-secure MPC protocol with communication complexity $O(n)$ per multiplication is presented. The protocol is not designed in the pre-processing model, but rather in the player-elimination framework, where the circuit is divided into segments and each segment is evaluated “optimistically”, assuming no fault will occur. At the end of the segment evaluation, a detection protocol is executed to identify whether the segment is evaluated correctly and if any inconsistency is detected, then a fault-localization protocol is executed. The fault-localization process identifies a pair of parties, with at least one of them being corrupted. The pair is then neglected for the rest of the protocol execution and the procedure is repeated. There are several drawbacks of this protocol. The protocol cannot be adapted in the pre-processing model, so the benefits provided by the pre-processing based MPC protocols (namely efficiently generating circuit-independent raw materials for several instances of the computation in parallel) cannot be
obtained by it. The protocol also makes expensive use of zero-knowledge (ZK) machinery throughout the protocol and it does not seem to be adaptable to the asynchronous setting with \( \mathcal{O}(n) \) communication complexity. Our techniques on the other hand are focused on efficient protocols in the pre-processing model, for example we use ZK tools only in the offline phase, and our methods are easily applicable to the asynchronous setting.

Our Contribution: As remarked earlier we present a method to obtain \( \mathcal{O}(n) \) communication complexity (measuring in terms of bits and not just messages sent) for the online phase of robust MPC protocols with \( t < n/2 \). We are focused on protocols which could be practically relevant, so we are interested in suitable modifications of protocols such as VIFF \([14]\), BDOZ \([7]\) and SPDZ \([16]\). Our main contribution is a trick to robustly reconstruct a batch of \( \Theta(n(t + 1)) \) secret shared values with a communication complexity of \( \mathcal{O}(n^3) \) bits. Thus for \( t = \Theta(n) \) we obtain an amortized communication complexity of \( \mathcal{O}(n) \) bits for reconstructing a value. Assuming our arithmetic circuit is suitably wide, this implies an \( \mathcal{O}(n) \) online phase when combined with the standard method for evaluating multiplication gates based on pre-processed Beaver triples.

To produce this sub-protocol we utilize the error-correcting capability of the underlying secret-sharing scheme when error positions are already known. The error positions are however identified using the application of the pairwise BDOZ MACs from \([7]\). The overall sub-protocol is highly efficient and can be utilized in practical MPC protocols. Interestingly our reconstruction protocol also works in the asynchronous setting. Thus we obtain a practical optimization in both synchronous and asynchronous setting.

Since our optimization also works in the asynchronous setting, we end the paper by discussing how the offline phase, and the interaction between the offline and online phases can be handled in the asynchronous setting. In the VIFF framework \([14]\), which implements the offline phase with \( t < n/3 \) via the pseudo-random secret sharing, a single synchronization point is needed between the offline and online phases. Following the same approach, with some additional technicalities, we show how the interaction between offline and online phase can be handled asynchronously with \( t < n/2 \).

The SHE based offline phase of SPDZ can easily be adapted to the case of threshold secret sharing (via using threshold decryption techniques for SHE from \([1]\) or \([12]\)). The adaption to the offline phase to output BDOZ, as opposed to SPDZ, style MACs is also trivial. However, we know of no work which has examined this offline phase in the asynchronous setting. We present some partial results in this direction.

2 Preliminaries

We assume a set of \( n \) parties \( \mathcal{P} = \{ P_1, \ldots, P_n \} \), connected by pair-wise authentic channels, and a centralized static and active PPT adversary \( A \) who can corrupt any \( t < n/2 \) parties at the beginning of the execution of a protocol and can force them to behave in an arbitrary manner. For simplicity we assume \( n = 2t + 1 \), so that \( t = \Theta(n) \) holds. The functionality that the parties wish to compute will be represented by an arithmetic circuit over a finite field \( F \), where \( |F| > n \), and with \( \log |F| = \mathcal{O}(\kappa) \), where \( \kappa \) is the security parameter. We assume all computation and communication is performed over \( F \). A negligible function in \( \kappa \) will be denoted by \( \text{negl}(\kappa) \). For two \( m \) length vectors \( A = (a_1, \ldots, a_m) \) and \( B = (b_1, \ldots, b_m) \) over \( F \), we will denote by \( A \otimes B \) the value \( \sum_{i=1}^{m} a_i \cdot b_i \).

2.1 Communication

We will mainly consider two different communication settings. The first setting is the popular and simple, but less practical, synchronous channel setting, where the channels are synchronous and there is a strict upper bound on the message delays. All the parties in this setting are assumed to be synchronized via a global clock. Any protocol in this setting operates as a sequence of rounds, where in every round, it first performs some computation, then it sends messages to the parties over the pair-wise channels and broadcasts any message which needs to be broadcast to all the parties; this stage is followed by receiving the messages sent to the party by the other parties over the pair-wise channels and the messages broadcast by the parties in the previous round. Since the system is synchronous, any (honest) party need not have to wait endlessly for any message in any round. Thus the standard behaviour is to assume that if a party does not receive a value which it is supposed to receive or instead it receives a “syntactically incorrect” value, then the party simply substitutes a default value (instead of waiting endlessly, since the sender is definitely corrupted in this case) and proceeds further to the next round.
The other communication setting is the more involved, but more practical, asynchronous setting; here the channels are asynchronous and messages can be arbitrarily (but finitely) delayed. The only guarantee in this model is that the messages sent by the honest parties will eventually reach their destinations. The order of the message delivery is decided by a scheduler. To model the worst case scenario, we assume that the scheduler is under the control of the adversary. The scheduler can only schedule the messages exchanged between the honest parties, without having access to the “contents” of these messages. As in [5,9], we consider a protocol execution in this setting as a sequence of atomic steps, where a single party is active in each step. A party is activated when it receives a message. On receiving a message, it performs an internal computation and then possibly sends messages on its outgoing channels. The order of the atomic steps are controlled by the scheduler. At the beginning of the computation, each party will be in a special start state. A party is said to terminate/complete the computation if it reaches a halt state, after which it does not perform any further computation. A protocol execution is said to be complete if all the honest parties terminate the computation.

It is easy to see that the asynchronous setting models real-world networks like the Internet (where there can be arbitrary message delays) more appropriately than the synchronous setting. Unfortunately, designing protocol in the asynchronous setting is complicated and this stems from the fact that we cannot distinguish between a corrupted sender (who does not send any messages) and a slow but honest sender (whose messages are arbitrarily delayed). Due to this the following unavoidable but inherent phenomenon is always present in any asynchronous protocol: at any stage of the protocol, no (honest) party can afford to receive communication from all the n parties, as this may turn out to require an endless wait. So as soon as the party listens from n − t parties, it has to proceed to the next stage; but in this process, communication from t potentially honest parties may get ignored.

2.2 Primitives

Linearly-homomorphic Encryption Scheme (HE). For our efficient public reconstruction protocol, we assume an IND-CPA secure linearly-homomorphic public-key encryption scheme set-up for every party Pi ∈ P with message space F; a possible instantiation could be the BGV scheme [8]. Under this set-up, party Pi will own a secret decryption key dk(i) and the corresponding encryption key pk(i) will be publicly known. Given pk(i), a plaintext x and a randomness r, anyone can compute a ciphertext HE.\text{c}(x) \overset{def}{=} HE.\text{Enc}_{pk(i)}(x, r) of x for Pi, using the encryption algorithm HE.\text{Enc}, where the size of HE.\text{c}(x) is O(κ) bits. Given a ciphertext HE.\text{c}(x) = HE.\text{Enc}_{pk(i)}(x, *) and the decryption key dk(i), party Pi can recover the plaintext x = HE.\text{Dec}_{dk(i)}(c_x) using the decryption algorithm HE.\text{Dec}.

The encryption scheme is assumed to be linearly homomorphic: given two ciphertexts HE.\text{c}(x) = HE.\text{Enc}_{pk(i)}(x, *) and HE.\text{c}(y) = HE.\text{Enc}_{pk(i)}(y, *), there exists an operation, say ⊕, such that HE.\text{c}(x) ⊕ HE.\text{c}(y) = HE.\text{Enc}_{pk(i)}(x + y, *). Moreover, given a ciphertext HE.\text{c}(x) = HE.\text{Enc}_{pk(i)}(x, *) and a publicly known constant c, there exists some operation, say ⊙, such that c ⊙ HE.\text{c}(x) = HE.\text{Enc}_{pk(i)}(c \cdot x, *).

Information-theoretic MACs: In our protocols, we will use information-theoretically secure message authentication codes (MAC), similar to the one used in [7]. The authenticating key is a random variable in F and the MAC on a value a is its corresponding point on the line. Specifically, for such MACs, a random pair K = (α, β) ∈ F is selected as the MAC key and the MAC tag on a value a ∈ F, under the key K is defined as MACK(a) \overset{def}{=} α \cdot a + β. In our constructions, the MACs are used as follows: a party Pi will hold some value a and a MAC tag MAC(a), while another party Pj will hold the MAC key K. Later when Pi wants to disclose a to Pj, it sends a along with MAC(a); Pj verifies if a is consistent with the MAC tag with respect to its key K. A corrupted party Pj on holding the MAC tag on a message gets one point on the straight-line y = αx + β and it leaves one degree of freedom on the polynomial. Therefore Pj, even with unbounded computing power, cannot recover K completely. So a corrupted Pj cannot reveal an incorrect value α' ≠ a to an honest Pj without getting caught except with probability 1/\mathbb{F} = 2^{-κ} = \text{negl}(κ), which is the probability of guessing a second point on the straight-line and thus guessing the straight-line.

Definition 1 (Consistent MAC Keys). We call two MAC keys K = (α, β) and K' = (α', β') consistent if α = α'.

We define the following operations on MAC key(s)

- For two consistent MAC keys K = (α, β) and K' = (α', β'), we have K + K' \overset{def}{=} (α + α', β + β').
Given a value \( c \), then \( K + c \overset{def}{=} (\alpha, \beta + \alpha c) \) and \( K - c \overset{def}{=} (\alpha, \beta - \alpha c) \).

Given a value \( c \), then \( c \cdot K \overset{def}{=} (\alpha, c \cdot \beta) \).

The MAC tags satisfy the linearity property in the sense that given two consistent MAC keys \( K, K' \) and a value \( c \), then the following holds (we leave the proof as an easy exercise).

- **Addition:** \( \text{MAC}_K(a) + \text{MAC}_{K'}(b) = \text{MAC}_{K+K'}(a + b) \).
- **Addition/Subtraction by a Constant:** \( \text{MAC}_{K-c}(a + c) = \text{MAC}_K(a) \) and \( \text{MAC}_{K+c}(a - c) = \text{MAC}_K(a) \).
- **Multiplication by a constant:** \( c \cdot \text{MAC}_K(a) = \text{MAC}_{c \cdot K}(c \cdot a) \).

### 2.3 The Various Sharings

We define two types of secret sharing, the \([\cdot]\)-sharing and \(\langle\cdot\rangle\)-sharing, where the latter uses MACs.

**Definition 2** \(([\cdot]\text{-sharing}). We say a value \( s \in F \) is \([\cdot]\text{-shared among } P \) if there exists a polynomial \( p(\cdot) \) of degree at most \( t \) with \( p(0) = s \) and every (honest) party \( P_i \in P \) holds a share \( s_i \overset{def}{=} p(i) \) of \( s \). We denote by \([s] \) the vector of shares of \( s \) corresponding to the (honest) parties in \( P \). That is, \([s] = \{s_i\}_{i=1}^n \).

**Definition 3** \((\langle\cdot\rangle\text{-sharing}). We say that a value \( s \in F \) is \(\langle\cdot\rangle\text{-shared among } P \) if \( s \) is \([\cdot]\text{-shared among } P \) and every (honest) party \( P_i \) holds a MAC tag on its share \( s_i \) for a key \( K_{ji} \) held by every \( P_j \). That is, the following holds for every pair of (honest) parties \( P_i, P_j \in P \): party \( P_i \) holds MAC tag \( \text{MAC}_{K_{ji}}(s_i) \) for a MAC key \( K_{ji} \) held by \( P_j \). We denote by \( \langle s \rangle \) the vector of shares, MAC keys and MAC tags of \( s \) corresponding to the (honest) parties in \( P \). That is, \( \langle s \rangle = \{s_i, \{\text{MAC}_{K_{ji}}(s_i), K_{ji}\}_{j=1}^n\}_{i=1}^n \).

While most of our computation are done over values that are \(\langle\cdot\rangle\text{-shared}, our efficient public reconstruction protocol for \(\langle\cdot\rangle\text{-shared values will additionally require a tweaked version of \(\langle\cdot\rangle\text{-sharing}, where there exists some designated party, say } P_j; \text{ and the parties hold the shares and the MAC tags in an encrypted form under the public key } \text{pk}(j) \text{ of } P_j \text{ of an HE scheme, with } P_j \text{ knowing the corresponding secret key } \text{dk}(j). \text{ We stress that the shares and MAC tags will not be available in clear, but rather in an encrypted form. More formally:}

**Definition 4** \((\langle\langle\cdot\rangle\rangle)\text{-sharing}). Let \( s \in F \) and \([s] = \{s_i\}_{i=1}^n \) be the vector of shares corresponding to an \([\cdot]\text{-sharing of } s \). We say that \( s \) is \(\langle\langle\cdot\rangle\rangle\text{-shared among } P \) with respect to a designated party \( P_j \), if every (honest) party \( P_i \) holds an encrypted share \( \text{HE.c}(s_i) \) and encrypted MAC tag \( \text{HE.c(MAC}_{K_{ji}}(s_i)) \) under the public key \( \text{pk}(j) \), such that \( P_j \) holds the MAC keys \( K_{ji} \) and the secret key \( \text{dk}(j) \). We denote by \( \langle\langle s \rangle\rangle_j \) the vector of encrypted shares and encrypted MAC tags corresponding to the (honest) parties in \( P \), along with the MAC keys and the secret key of \( P_j \). That is, \( \langle\langle s \rangle\rangle_j = \{\text{HE.c}(s_i), \text{HE.c(MAC}_{K_{ji}}(s_i))\}_{i=1}^n, \{K_{ji}\}_{j=1}^n, \text{dk}(j)\} \).

**Private Reconstruction of \(\langle\cdot\rangle\text{- and } \langle\langle\cdot\rangle\rangle\text{-shared Value Towards a Designated Party.}** Note that with \( n = 2t+1 \), a \([\cdot]\text{-shared value cannot be robustly reconstructed towards a designated party just by sending the shares, as we cannot do error-correction. However, we can robustly reconstruct a \(\langle\cdot\rangle\text{-sharing towards a designated party, say } P_j, \text{ by asking the parties to send their shares, along with the MAC tags to } P_j, \text{ who then identifies the correct shares with high probability and reconstructs the secret. A similar idea can be used to reconstruct an } \langle\langle s \rangle\rangle_j \text{-sharing towards } P_j \text{. Now the parties send encrypted shares and MAC tags to } P_j, \text{ who decrypts them before doing the verification. We call the resultant protocols } \text{RecPrv}^j(\langle s \rangle, P_j) \text{ and } \text{RecPrvEnc}^j(\langle\langle s \rangle\rangle_j, P_j) \text{ respectively, which are presented in Fig. 1. We stress that while } \langle s \rangle \text{ can be reconstructed towards any } P_j, \text{ } \langle\langle s \rangle\rangle_j \text{ can be reconstructed only towards } P_j, \text{ as } P_j \text{ alone holds the secret key } \text{dk}(j) \text{ that is required to decrypt the shares and the MAC tags.}

It is easy to see that if \( P_j \) is honest, then \( P_j \) correctly reconstructs the shared value in both the protocols, except with probability at most \( \frac{1}{2t} \overset{negl}{=} \) in the asynchronous setting. We argue this for \( \text{RecPrv} \) (the same argument will work for \( \text{RecPrvEnc} \)). The party \( P_j \) will eventually receive the shares of \( s \) from at least \( n-t = t+1 \) honest parties, with correct MACs. These \( t+1 \) shares are enough for the robust reconstruction of \( s \). So we state the following lemma for \( \text{RecPrv}. \) Similar statements hold for protocol \( \text{RecPrvEnc}. \)
Lemma 1. Let $s$ be a $\langle \cdot \rangle$-shared among the parties $\mathcal{P}$. Let $P_j$ be a specific party. Protocol RecPrv achieves the following in the synchronous communication setting:

- Correctness: Except with probability $\text{negl}(\kappa)$, an honest $P_j$ reconstructs the value $s$.
- Communication Complexity: The required communication complexity is $O(\kappa \cdot n)$ bits.

Lemma 2. Let $s$ be a $\langle \cdot \rangle$-shared among the parties $\mathcal{P}$. Let $P_j$ be a specific party. Protocol RecPrv achieves the following in the asynchronous communication setting:

- Correctness & Communication Complexity: Same as in Lemma 1
- Termination: If every honest party participates in RecPrv, then an honest $P_j$ will eventually terminate.

Linearity of Various Sharings. All of the previously defined secret sharings are linear, which for ease of exposition we shall now overview. We first define what is meant by key consistent sharings.

Definition 5 (Key-consistent $\langle \cdot \rangle$ and $\langle \langle \cdot \rangle \rangle_j$-sharings). Two $\langle \cdot \rangle$-sharings $\langle a \rangle$ and $\langle b \rangle$ are said to be key-consistent if every (honest) $P_i$ holds consistent MAC keys for every $P_j$ across both the sharings.

Two $\langle \langle \cdot \rangle \rangle_j$-sharings $\langle \langle a \rangle \rangle_j$ and $\langle \langle b \rangle \rangle_j$ with respect to a designated party $P_j$ are said to be key-consistent if (honest) $P_j$ holds consistent MAC keys for every $P_i$ across both the sharings, and the encryptions are under the same public key of $P_j$.

Linearity of $\langle \cdot \rangle$-sharings: Given sharings $[a] = \{a_i\}_{i=1}^n$ and $[b] = \{b_i\}_{i=1}^n$ and a publicly known constant $c$, we have:

- Addition: $[a] + [b] = [a+b] = \{a_i + b_i\}_{i=1}^n$. To compute $[a+b]$, every party $P_i$ needs to locally compute $a_i + b_i$.
- Addition by a Public Constant: $c + [a] = [c + a] = \{c + a_i\}_{i=1}^n$. To compute $[c + a]$, every party $P_i$ needs to locally compute $c + a_i$.
- Multiplication by a Public Constant: $c \cdot [a] = \{c \cdot a_i\}_{i=1}^n$. To compute $[c \cdot a]$, every party $P_i$ needs to locally compute $c \cdot a_i$.

Linearity of $\langle \langle \cdot \rangle \rangle_j$-sharings: Given sharings $\langle a \rangle = \{\text{MAC}_{K_{ij}}(a_i), K_{ij}\}_{j=1}^n$ and $\langle b \rangle = \{b_i, \text{MAC}_{K'_{ij}}(b_i), K'_{ij}\}_{j=1}^n$, that are key-consistent and a publicly-known constant $c$, we have:

- Addition: $\langle a \rangle + \langle b \rangle = \langle a + b \rangle = \{a_i + b_i, \text{MAC}_{K_{ij}+K'_{ij}}(a_i + b_i), K_{ij} + K'_{ij}\}_{j=1}^n$. To compute $\langle a + b \rangle$, every party $P_i$ needs to locally compute $a_i + b_i$, $\{\text{MAC}_{K_{ij}}(a_i) + \text{MAC}_{K'_{ij}}(b_i)\}_{j=1}^n$ and $\{K_{ij} + K'_{ij}\}_{j=1}^n$.
- Addition by a Public Constant: $c + \langle a \rangle = \langle c + a \rangle = \{c + a_i, \text{MAC}_{K_{ij}}(a_i + c), K_{ij} + c\}_{j=1}^n$. To compute $\langle c + a \rangle$, every party $P_i$ needs to locally compute $c + a_i$. We recall that $\text{MAC}_{K_{ij}}(a_i + c) = \text{MAC}_{K_{ij}}(a_i + c)$. Hence we assign $\text{MAC}_{K_{ij}}(a_i)$ to $\text{MAC}_{K_{ij},-c}(a_i + c)$ and compute $\{K_{ij} + c\}_{j=1}^n$. 

Fig. 1. Protocols for Reconstructing a $\langle \cdot \rangle$-sharing and $\langle \langle \cdot \rangle \rangle_j$-sharing Towards a Designated Party.
Linearity of $\langle\cdot\rangle_j$-sharings: Given
$$\langle(a)\rangle_j = \\{\{\text{HE.c}(a_i), \text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(a_i)))\}_{i=1}^n, \{K_{ji}\}_{i=1}^n, \text{dk}(j)\}$$
$$\langle(b)\rangle_j = \\{\{\text{HE.c}(b_i), \text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(b_i)))\}_{i=1}^n, \{K'_{ji}\}_{i=1}^n, \text{dk}(j)\}$$

that are key-consistent we can add the ciphertexts via the operation
$$\langle(a)\rangle_j + \langle(b)\rangle_j = \langle(a + b)\rangle_j = \\{\{\text{HE.c}(a_i + b_i), \text{HE.c}((\text{MAC}_{\mathcal{K}_{ji} + K'_{ji}}(a_i + b_i)))\}_{i=1}^n, \{K_{ji} + K'_{ji}\}_{i=1}^n, \text{dk}(j)\}.$$

So to compute $\langle(a + b)\rangle_j$, every party $P_i \in \mathcal{P}$ needs to locally compute the values $\text{HE.c}(a_i) \oplus \text{HE.c}(b_i)$ and $\text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(a_i)) \oplus \text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(b_i)))$, while party $P_j$ needs to compute $\{K_{ji} + K'_{ji}\}_{i=1}^n$. Multiplication by public constants for key-consistent $\langle\cdot\rangle_j$-sharings can also be done. However we skip the details as we need only the above linearity property from $\langle\cdot\rangle_j$-sharings in our efficient public reconstruction protocol.

Generating $\langle\cdot\rangle_j$-sharing from $\langle\cdot\rangle$-sharing. In our efficient protocol for public reconstruction of $\langle\cdot\rangle$-shared values, we come across the situation where there exists: a value $r$ known only to a designated party $P_j$, a publicly known encryption $\text{HE.c}(r)$ of $r$, under the public key $\text{pk}^{(j)}$, and a $\langle\cdot\rangle$-sharing $\langle a \rangle = \{a_i, \{\text{MAC}_{\mathcal{K}_{ji}}(a_i), K_{ji}\}_{i=1}^n\}$. Given the above, the parties need to compute a $\langle\cdot\rangle_j$-sharing:
$$\langle(r \cdot a)\rangle_j = \\{\{\text{HE.c}(r \cdot a_i), \text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(r \cdot a_i)))\}_{i=1}^n, \{r \cdot K_{ji}\}_{i=1}^n, \text{dk}(j)\}$$
of $r \cdot a$. Computing above needs only local computation by the parties. Specifically, each party $P_i \in \mathcal{P}$ locally computes:
$$\text{HE.c}(r \cdot a_i) = a_i \oplus \text{HE.c}(r)$$
$$\text{HE.c}((\text{MAC}_{\mathcal{K}_{ji}}(r \cdot a_i))) = \text{HE.c}((r \cdot \text{MAC}_{\mathcal{K}_{ji}}(a_i)))$$
$$\text{MAC}_{\mathcal{K}_{ji}}(r \cdot a_i) = \text{MAC}_{\mathcal{K}_{ji}}(a_i) \oplus \text{HE.c}(r)$$
Finally party $P_j$ locally computes $\{r \cdot K_{ji}\}_{i=1}^n$.

3 Efficient Public Reconstruction of $\langle\cdot\rangle$-sharings with a Linear Overhead

In this section, we show how to publicly reconstruct $n(t+1) = \Theta(n^2)$ $\langle\cdot\rangle$-shared values with communication complexity $O(n \cdot t^3)$ bits. This suggests a method for public reconstruction of one $\langle\cdot\rangle$-shared value with an amortized communication overhead that is linear in the number of parties i.e. $O(n \cdot n^2)$ bits.

Let $\{(a^{(i,j)})\}_{i=1, j=1}^{n, t+1}$ be the $\langle\cdot\rangle$-sharings, which need to be publicly reconstructed. The naive way of achieving the task is to run $\Theta(n^3)$ instances of RecPur, where $\Theta(n^2)$ instances are run to reconstruct all the values to a single party. This method has communication complexity $O(n \cdot n^4)$ bits and thus have a quadratic overhead. Our approach outperforms the naive method. Our protocol, called RecPub works for both synchronous as well as asynchronous setting; for simplicity we first explain the protocol assuming a synchronous setting.

Let $A$ be an $n \times (t+1)$ matrix, with $(i, j)$th element as $a^{(i,j)}$. Let $A_i(x)$ be a polynomial of degree $t$ defined over the values in the $i$th row of $A$; i.e., $A_i(x) \equiv a^{(i,1)} + a^{(i,2)}x + \ldots + a^{(i,t+1)}x^t$. Let $b^{(i,j)} \equiv A_i(j)$, for $i, j \in \{1, \ldots, n\}$ and let $B$ denote the $n \times n$ matrix with $(i, j)$th element as $b^{(i,j)}$. Clearly $A$ can be recovered given any $t+1$ columns of $B$. We explain below how to reconstruct at least $t+1$ columns of $B$ to all the parties with communication complexity $O(n \cdot n^3)$ bits. In what follows, we denote $i$th row and column of $A$ as $A_i$ and $A^j$ respectively, with a similar notation used for the rows and columns of $B$. 

6
Since every $B_i$ is linearly dependent on $A_i$, given $(\cdot)$-sharing of the elements in $A_i$, it requires only local computation by the parties to generate $(\cdot)$-sharing of the elements in $B_i$. Specifically, $(a^{(i,j)}) = (a^{(i,1)}) + (a^{(i,2)}) \cdot j + \ldots + \langle a^{(i,t+1)} \rangle \cdot j^t$. Then we reconstruct the elements of $A$ to all the parties in two steps. First $B^i$ is reconstructed only towards party $P_i$ using $n$ instances of RecPrv with an overall cost $O(\kappa \cdot n^3)$ bits. Next each party $P_i$ sends $B^i$ to all the parties, requiring $O(\kappa \cdot n^3)$ bits of communication. If every $P_i$ behaves honestly then every party would possess $B$ at the end of the second step. However a corrupted $P_i$ may not send the correct $B^i$. So what we need is a mechanism that allows an honest party to detect if a corrupted party $P_i$ has sent an incorrect $B^i$. Detecting is enough, since every (honest) party is guaranteed to receive correctly the $B^i$ columns from $t + 1$ honest parties. Recall that $t + 1$ correct columns of $B$ are enough to reconstruct $A$.

After $P_i$ reconstructs $B^i$, and before it sends the same to party $P_j$, we allow $P_j$ to obtain a random linear combination of the elements in $B^i$ (via interaction) in a way that the linear combiners are known to no one other than $P_j$. Later, when $P_i$ sends $B^i$ to $P_j$, party $P_j$ can verify if the $B^i$ received from $P_i$ is correct or not by comparing the linear combination of the elements of the received $B^i$ with the linear combination that it obtained before. It is crucial to pick the linear combiners randomly and keep them secret, otherwise $P_i$ can cheat with an incorrect $B^i$ without getting detected by an honest $P_j$. In our method, the random combiners for an honest $P_j$ are never leaked to anyone and this allows $P_j$ to reuse them in a latter instance of the public reconstruction protocol. Specifically, we assume the following one time setup for RecPub (which can be done beforehand in the offline phase once and for all). Every party $P_i$ holds a secret key $dk^{(j)}$ for the linearly-homomorphic encryption scheme $\text{HE}$ and the corresponding public key $pk^{(j)}$ is publicly available. In addition, $P_i$ holds $n$ random combiners, say $R^{(j)} = (r^{(j,1)}, \ldots, r^{(j,n)})$, and the encryptions $\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})$ of these values under $P_j$’s public key $pk^{(j)}$ are available publicly. The above setup can be created once and for all, and can be reused across multiple instances of RecPub executed either within one instance of the function evaluation or spread across many instances of function evaluations.

Given the above random combiners in an encrypted form, party $P_j$ can obtain the linear combination $c^{(i,j)} \overset{def}{=} r^{(j,1)}b^{(1,i)} + \ldots + r^{(j,n)}b^{(n,i)}$ of the elements of $B^i$ as follows. First note that the parties hold $(\cdot)$-sharing of the elements of $B^i$. If the linear combiners were publicly known, then the parties could compute $(c^{(i,j)}) = r^{(j,1)}(b^{(1,i)}) + \ldots + r^{(j,n)}(b^{(n,i)})$ and reconstruct $c^{(i,j)}$ to party $P_j$ using RecPrv. However since we do not want to disclose the combiners, the above task is performed in an encrypted form, which is doable since the combiners are encrypted under the linearly-homomorphic encryption scheme $\text{HE}$. Specifically, given encryptions $\text{HE}.c(r^{(j,k)})$ under $pk^{(j)}$ and sharings $\langle b^{(k,i)} \rangle$, the parties first generate $\langle \langle r^{(j,k)} : b^{(k,i)} \rangle \rangle_j$ for every $P_j$ (recall that it requires only local computation). Next the parties linearly combine the sharings $\langle \langle r^{(j,k)} : b^{(k,i)} \rangle \rangle_j$ for $k = 1, \ldots, n$ to get $\langle \langle c^{(i,j)} \rangle \rangle_j$, which is then reconstructed towards party $P_j$ using an instance of RecPrvEnc. In total $n^2$ such instances need to be executed and so the communication cost of the above step is $O(\kappa \cdot n^3)$ bits. Protocol RecPub is now formally presented in Fig. 2.

The correctness and communication complexity of the protocol are stated in Lemma 3, which follows in a straight-forward fashion from the protocol description and the detailed protocol overview. The security of the protocol will be proven in conjunction with the online phase of our MPC protocol in the next section.

**Lemma 3.** Let $\{\langle a^{(i,j)} \rangle\}_{i=1,j=1}^{n,t+1}$ be a set of $n(t + 1) = \Theta(n^2)$ shared values which need to be publicly reconstructed by the parties. Then given a setup $(pk^{(1)}, dk^{(1)}), \ldots, (pk^{(n)}, dk^{(n)})$ for the linearly-homomorphic encryption scheme $\text{HE}$ for the $n$ parties and encryptions $\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})$ of $n$ random values $r^{(j,1)}, \ldots, r^{(j,n)}$ on the behalf of each party $P_j \in P$, with only $P_j$ knowing the random values, protocol RecPub achieves the following in the synchronous communication setting:

- Correctness: Except with probability $\text{negl}(\kappa)$, every honest party reconstructs the values $\{a^{(i,j)}\}_{i=1,j=1}^{n,t+1}$.
- Communication Complexity: The protocol has communication complexity $O(\kappa \cdot n^3)$ bits.

**Protocol RecPub in the Asynchronous Setting:** A closer look at the protocol RecPub reveals that the protocol works in an asynchronous setting. More specifically, assume that every honest party participates in the protocol RecPub, then we can show that every honest party will terminate the protocol. Transforming the sharings of the values in the matrix $A$ into the sharings of the values in matrix $B$ requires only local computation which will be eventually completed by the honest parties. Next the instances of RecPrv and RecPrvEnc for each honest $P_j$ will terminate by Lemma 2. So every honest $P_j$ will eventually reconstruct $B^i$ and $c^{(j,i)}$ (the latter value is reconstructed irrespective of an honest or corrupted $P_j$). Note that the steps between the reconstruction of $B^i$ and $c^{(j,i)}$ involves only local computation. Finally,
Protocol RecPub(\(\{(a^{(i,j)})\}_{i=1,j=1}^{n,t+1}\))

Each party \(P_i \in \mathcal{P}\) holds \(R^j = (r^{(j,1)}, \ldots, r^{(j,n)})\) and the encryptions \(\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})\), under \(P_j\)’s public key \(\text{pk}^{(j)}\), are publicly known. Let \(A\) be the matrix of size \(n \times (t+1)\), with the \((i,j)\)th entry as \(a^{(i,j)}\), for \(i \in \{1, \ldots, n\}\) and \(j \in \{1, \ldots, t+1\}\). We denote the \(i\)th row and column of \(A\) as \(A_i\) and \(A_j^t\) respectively. We define \(A_{ik}(x) \overset{def}{=} a^{(i,1)} + a^{(i,2)}x + \ldots + a^{(i,t+1)}x^t\) for \(i \in \{1, \ldots, n\}\), and \(b^{(i,j)} \overset{def}{=} A_i(j)\) for \(i, j \in \{1, \ldots, n\}\). Let \(B\) be the matrix of size \(n \times n\), with the \((i,j)\)th entry as \(b^{(i,j)}\). We denote the \(i\)th row and column of \(B\) as \(B_i\) and \(B^j\) respectively. The parties do the following to reconstruct \(A\):

- Computing \((\cdot)\)-sharing of every element of \(B\): For \(i, j \in \{1, \ldots, n\}\), the parties compute \(\langle b^{(i,j)} \rangle = \langle a^{(i,1)} \rangle + j \cdot \langle a^{(i,t+1)} \rangle\).
- Reconstructing \(B^j\) towards \(P_i\): For \(i \in \{1, \ldots, n\}\), the parties execute \(\text{RecPrv}(\langle b^{(1,1)} \rangle, P_1), \ldots, \text{RecPrv}(\langle b^{(n,n)} \rangle, P_1)\) to enable \(P_i\) robustly reconstruct \(B^i\).
- Reconstructing \(B^i \otimes R^j\) towards \(P_j\): Corresponding to each \(P_i \in \mathcal{P}\), the parties execute the following steps, to enable each \(P_j \in \mathcal{P}\) to obtain the random linear combination \(c^{(i,j)} \overset{def}{=} B^i \otimes R^j = r^{(j,1)}b^{(1,i)} + \ldots + r^{(j,n)}b^{(n,i)}\):
  - The parties first compute \(\langle (r^{(j,k)} \cdot b^{(k,i)}) \rangle_j\) from \(\text{HE}.c(r^{(j,k)})\) and \(\langle b^{(k,i)} \rangle\) for \(k \in \{1, \ldots, n\}\) and then compute \(\langle (c^{(i,j)}) \rangle = \sum_{k=1}^n \langle (r^{(j,k)} \cdot b^{(k,i)}) \rangle\).
  - The parties execute \(\text{RecPrvEnc}(\langle (c^{(i,j)}) \rangle_j)\) to reconstruct \(c^{(i,j)}\) towards \(P_j\).
- Sending \(B^j\) to all: Every party \(P_i \in \mathcal{P}\) sends \(B^i\) to every party \(P_j \in \mathcal{P}\). Each party \(P_j\) then reconstructs \(A\) as follows:
  - On receiving \(B^j\) from \(P_i\), compute \(c^{(i,j)} = B^i \otimes R^j\) and check if \(c^{(i,j)} = c^{(i,j)}\). If the test passes then \(P_j\) considers \(B^j\) as the valid \(i\)th column of the matrix \(B\).
  - Once \(t+1\) valid columns of \(B\) are obtained by \(P_j\), it then reconstructs \(A\).

**Fig. 2.** Robustly Reconstructing \((\cdot)\)-shared Values with a Linear Overhead

---

4 Linear Overhead Online Phase Protocol

Let \(f\) be a publicly known function over \(\mathbb{F}\) which takes a single input from each party and has a single output specifically \(f : \mathbb{F}^n \rightarrow \mathbb{F}\). Let \(f\) be represented as an arithmetic circuit \(C\) over \(\mathbb{F}\), consisting of \(M\) multiplication gates. We show how our efficient reconstruction protocol RecPub, enables one to securely realize the standard ideal functionality for the MPC evaluation of the circuit \(C\), \(\mathcal{F}_f\) (see Appendix A for an explicit functionality) in the \(\mathcal{F}_{\text{PREP}}\)-hybrid model, with communication complexity \(\mathcal{O}(\kappa \cdot (n \cdot M + n^2))\) bits, thus providing a linear overhead per multiplication gate. More specifically, assume that the parties have access to the pre-processing and input processing functionality \(\mathcal{F}_{\text{PREP}}\): This functionality creates the following one-time setup: (i) Every party \(P_i\) holds a secret key \(\text{dk}^{(j)}\) for the linearly-homomorphic encryption scheme \(\text{HE}\) and the corresponding public key \(\text{pk}^{(j)}\) is available publicly. In addition, each (honest) \(P_i\) holds \(n\) random combiners, say \(R^{(j)} = (r^{(j,1)}, \ldots, r^{(j,n)})\) and the encryptions \(\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})\) of these values under \(P_j\)’s public key are available with all the parties. (ii) Each party \(P_i\) holds \(\alpha_{ij}\), the \(\alpha\)-component of all its keys for party \(P_j\) (recall that for key-consistent sharings every \(P_i\) has to use the same \(\alpha\)-component for all its corresponding to \(P_j\)). The above setup can be reused across multiple instances of \(\Pi_{\text{ONLINE}}\) and can be created once and for all. In addition to the one-time setup, the functionality also helps create at least \(M\) random \((\cdot)\)-shared multiplications triples (these are not reusable and have to be created afresh for every execution of \(\Pi_{\text{ONLINE}}\)) and \((\cdot)\)-shared inputs of the parties. We provide the formal details of \(\mathcal{F}_{\text{PREP}}\) in Fig. 5 of Appendix A

---

4 This is just for simplicity; using standard techniques we can deal with functions taking multiple inputs from each party.

---

8
Using $\mathcal{F}_{\text{PREP}}$, we can design a protocol $\Pi_{\text{ONLINE}}$ (see Fig. 3) which UC-securely realizes $\mathcal{F}_f$ in the synchronous setting. The protocol is based on the standard idea of doing shared circuit evaluation based on pre-processed multiplication triples. A brief overview of the protocol is given in Appendix B. To achieve the linear overhead in $\Pi_{\text{ONLINE}}$, we require that the circuit is "wide" in the sense that at every level there are at least $n \cdot (t + 1)$ independent multiplication gates that can be evaluated in parallel. This is to ensure that we can use our linear-overhead reconstruction protocol $\text{RecPub}$. We note that similar restrictions are used in some of the previous MPC protocols to achieve a linear overhead in the online phase (see for example \cite{cryptoeprint:2011:015}). In practice many functions have such a level of parallelism gates when expressed in arithmetic circuit format, and practical systems use algorithms to maximise the level of such parallelism in their execution, see e.g. \cite{cryptoeprint:2020:1008}. Protocol $\Pi_{\text{ONLINE}}$ works for both the synchronous as well as asynchronous setting, but, for simplicity, we initially give the protocol for the synchronous setting and later argue that it works in the asynchronous setting.

**Protocol $\Pi_{\text{ONLINE}}$**

Every party $P_i \in \mathcal{P}$ interact with $\mathcal{F}_{\text{PREP}}$ with input Setup, Triples and $(x^{(i)}, \text{Input})$ and receives $d_{k^{(i)}}$, $\{pk^{(i)}\}_{j=1}^{n}$, $r_{k^{(i)}} = (r^{(i,1)}, \ldots, r^{(i,n)})$, $\{\text{HE}.c_{r^{(i,j,1)}}, \ldots, \text{HE}.c_{r^{(i,j,n)}}\}_{j=1}^{n}$, its information for multiplication triples $\{(a^{(i)}), (b^{(i)}), (c^{(i)})\}_{j=1}^{M}$ and its information for inputs $\{(x^{(i,j)})\}_{j=1}^{n}$. The honest parties associate the sharing $\{(a^{(i)}), (b^{(i)}), (c^{(i)})\}$ with the $t^\text{th}$ multiplication gate for $\mathcal{F}_f$ and evaluate each gate in the circuit as follows:

- **Linear Gates**: using the linearity property of $\langle \cdot \rangle$-sharing, the parties apply the linear function associated with the gate on the corresponding $\langle \cdot \rangle$-shared gate inputs to obtain a $\langle \cdot \rangle$-sharing of the gate output.

- **Multiplication Gates**: $M$ multiplication gates as grouped as a batch of $n \cdot (t + 1)$. We explain the evaluation for one batch.

  Let the inputs to the $r$th batch be $\{(p^{(i)}), (q^{(i)})\}_{i=1}^{n \cdot (t + 1)}$ and let $\{(a^{(i)}), (b^{(i)}), (c^{(i)})\}_{i=1}^{n \cdot (t + 1)}$ be the corresponding associated multiplication triples. To compute $\langle p^{(i)} \cdot q^{(i)} \rangle$, the parties do the following:

  - Locally compute $\langle d^{(i)} \rangle = \langle p^{(i)} \rangle - \langle a^{(i)} \rangle = \langle p^{(i)} \rangle - a^{(i)}$ and $\langle e^{(i)} \rangle = \langle q^{(i)} \rangle - b^{(i)}$.

  - Publicly reconstruct the values $\{d^{(i)}\}_{i=1}^{n \cdot (t + 1)}$ and $\{e^{(i)}\}_{i=1}^{n \cdot (t + 1)}$ using two instances of $\text{RecPub}$.

  - On reconstructing $\langle d^{(i)} \rangle, \langle e^{(i)} \rangle$, the parties set $\langle p^{(i)} \cdot q^{(i)} \rangle = \langle d^{(i)} \rangle \cdot \langle e^{(i)} \rangle + \langle b^{(i)} \rangle + \langle c^{(i)} \rangle - \langle a^{(i)} \rangle - \langle e^{(i)} \rangle$.

- **Output Gate**: Let $(y)$ be the sharing of the output gate. The parties execute $\text{RecPrv}(y, P_i)$ for every $P_i \in \mathcal{P}$, robustly reconstruct the function output $y$ and terminate.

**Fig. 3.** Protocol for Realizing $\mathcal{F}_f$ with a Linear Overhead in the $\mathcal{F}_{\text{PREP}}$-hybrid Model for the Synchronous Setting

The properties of $\Pi_{\text{ONLINE}}$ are stated in Theorem 1. We prove the security in the UC setting in Appendix B. The main communication in the protocol is for publicly reconstructing $\langle \cdot \rangle$-shared values while evaluating the multiplication gates; in total $2M$ such shared values need to be publicly reconstructed. We reconstruct them using $\text{RecPub}$. So assuming that the $M$ multiplication gates in the circuit $C$ can be divided into blocks of $n \cdot (t + 1)$ independent multiplication gates, evaluating the same via the Beaver’s trick will cost

$$O \left( \kappa \cdot n^3 \cdot \frac{M}{n \cdot (t + 1)} \right) = O(\kappa \cdot n \cdot M)$$

**Theorem 1.** Protocol $\Pi_{\text{ONLINE}}$ UC-securely realizes the functionality $\mathcal{F}_f$ in the $\mathcal{F}_{\text{PREP}}$-hybrid model in the synchronous setting. The protocol has communication complexity $O(\kappa \cdot (n \cdot M + n^2))$ bits.

**Protocol $\Pi_{\text{ONLINE}}$ in the Asynchronous Setting:** It is easy to see that protocol $\Pi_{\text{ONLINE}}$ will work in an asynchronous setting. That is, if every honest party participates in the protocol, then every honest party will eventually terminate the protocol. The steps in the protocol that involves interaction among the parties are the public reconstruction of values in the Beaver’s trick and the public reconstruction of the circuit output. By Lemma 4 if the honest parties participate in an instance of $\text{RecPub}$, then they will eventually terminate the execution. Furthermore, the instances of $\text{RecPrv}$ for reconstructing the circuit output will be eventually terminated for every honest party (by Lemma 2). The remaining steps in $\Pi_{\text{ONLINE}}$ involve only local computation which the honest parties will complete eventually. Hence we have the following theorem:
Theorem 2. Protocol $\Pi_{\text{ONLINE}}$ UC-securely realizes the functionality $\mathcal{F}_f$ in the $\mathcal{F}_{\text{PREP}}$-hybrid model in the asynchronous setting. The protocol has communication complexity $O(\kappa \cdot (n \cdot M + n^3))$ bits.

5 The Various Secure Realizations of $\mathcal{F}_{\text{PREP}}$

Securely Realizing $\mathcal{F}_{\text{PREP}}$ in the Synchronous Setting. We present a protocol $\Pi_{\text{PREP}}$ which UC-securely realizes $\mathcal{F}_{\text{PREP}}$ in the synchronous setting. The protocol is a straight forward adaptation of the offline phase protocol of SPDZ and BDOZ to deal with Shamir sharing, instead of additive sharing. Here we just provide the high level overview of $\Pi_{\text{PREP}}$ and refer to Appendix [C] for the complete formal details. In $\Pi_{\text{PREP}}$, we assume an ideal functionality $\mathcal{F}_{\text{ONE-TIME-SETUP}}$, which provides the following one-time set-up for the parties: (1) Set-up for a threshold somewhat homomorphic encryption (SHE) scheme; (2) Set-up for a public-key, secret-key pair of a linearly-homomorphic encryption scheme for each party; (3) Set-up for encrypted random combiners for the parties and (4) Set-up for the encrypted $\alpha$-components of MAC keys for the parties (for details, see Figure [7] in Appendix [C]. We stress that the functionality needs to be called only once and the set-up done by it can be re-used across multiple instances of $\Pi_{\text{PREP}}$. For both SHE as well as HE, we assume the availability of standard UC-secure ZK protocols for proof of plain-text knowledge (PoPK); additionally for SHE, we assume UC-secure ZK protocols for proof of correct decryption (PoCD).

We explain how one random shared multiplication triple $\langle (a), (b), (c) \rangle$ is generated. The parties first generate $\langle (a_1), (b), (c) \rangle$: to generate $a_1$, we ask $t + 1$ parties to define a polynomial of degree at most $t$ in a shared fashion, where each party contributes one random point on the polynomial. For this, each party $P_i \in \{P_0, \ldots, P_{t+1}\}$ selects a random $a_i$, encrypts it using the SHE scheme and broadcasts the ciphertext with a ZK PoPK. Next, we define the polynomial $A(\cdot)$ of degree at most $t$, passing through the points $(1, a_1), \ldots, (t+1, a_{t+1})$. Since there exists at least one honest party in the set $\{P_1, \ldots, P_{t+1}\}$ whose corresponding $a_i$ value will be random and private, it follows that we will have one degree of freedom in the $A(\cdot)$ polynomial. So we can define $[a]$ via the polynomial $A(\cdot)$. To complete $[a]$, the remaining $t$ points on $A(\cdot)$ are computed homomorphically in an encrypted fashion and decrypted (in a distributed fashion) towards the corresponding parties. Following a similar procedure, the parties generate $[b]$ for a random $b$. At this stage, the parties will have an encryption of $a$ and $b$ under the SHE key and so they can homomorphically compute an encryption of $c$. To convert the encryption of $c$ into a $[-\cdot]$-sharing of $c$, we use the following standard idea used in SPDZ [16]: the parties generate an encryption of a random $r$, along with $[r]$. Then an encryption of $c + r$ is homomorphically computed and then publicly decrypted. Finally from $c + r$ and $[r]$, the parties obtain $[c]$. The generation of the MAC values follows in the standard BDOZ manner.

The major communication in the protocol is for broadcasting the encryptions and for distributed decryption. We assume that each party has to broadcast “sufficiently large” number of encryptions, so that we can use the multi-valued broadcast protocol of [13] with $O(n)$ overhead. For distributed decryption, we follow a protocol from [12] called DistDec (see Appendix [C]) which keeps the overall number of instances of PoCD during distributed decryption to $O(n^3)$. Since the protocol is a straight forward adaptation of the offline phase protocol of [7,16], we avoid giving the complete proof. Instead we state the following theorem.

Theorem 3. Protocol $\Pi_{\text{PREP}}$ UC-securely realizes $\mathcal{F}_{\text{PREP}}$ in the $\mathcal{F}_{\text{ONE-TIME-SETUP}}$-hybrid model in synchronous setting with communication complexity $O(\kappa \cdot (n^2 \cdot M + poly(n)))$ bits.

Securely Realizing $\mathcal{F}_{\text{PREP}}$ with Abort in the Partial Synchronous Setting. We now discuss how to securely realize $\mathcal{F}_{\text{PREP}}$ asynchronously. We first note that in a completely asynchronous setting, it is impossible to securely realize $\mathcal{F}_{\text{PREP}}$ in the point-to-point channel. This is because any secure realization of $\mathcal{F}_{\text{PREP}}$ has to ensure that all the honest parties have a “consistent” view of the protocol outcome. For this it is necessary that all the honest parties have an agreement on the final outcome. Unfortunately it is known that computationally secure asynchronous Byzantine agreement (ABA) is possible if and only if $t < n/3$ [21,22]. The second inherent difficulty in securely realizing $\mathcal{F}_{\text{PREP}}$ in an asynchronous setting is that it is impossible to ensure input provision from all the $n$ parties, as this may turn out to be endless. In the worst case, inputs of only $n - t$ parties can be considered for the computation and so for $n = 2t + 1$ this implies that out of the $t + 1$ input providers, there may be only one honest party. This may not be acceptable for most practical applications of MPC. To get rid of the latter difficulty, [14] introduced the following practically motivated variant of the traditional asynchronous communication setting, which we refer as partial asynchronous setting:
– The protocols in the partial asynchronous setting have one synchronization point. Specifically, there exists a certain well defined time-out and the assumption is that all the messages sent by the honest parties before the deadline will reach to their destinations within this deadline.

– Any protocol executed in the partial asynchronous setting need not always terminate and provide output to all the honest parties. Thus the adversary may cause the protocol to fail. However it is required that the protocol up to the synchronization point does not release any new information to the adversary.

In Appendix D we examine how to make $\Pi_{PRF}$ work in the partial asynchronous setting. We present two solutions; the first which allows some synchronous rounds after the synchronization point, and one which uses a method to enable a non-equivocation mechanism (which can be implemented using a trusted hardware module).

6 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO and EPSRC via grants EP/I03126X and EP/M016803.

References


A The $\mathcal{F}_f$ and $\mathcal{F}_{\text{PREP}}$ Functionalities

$\mathcal{F}_f$ is presented in Fig. 4, whilst $\mathcal{F}_{\text{PREP}}$ is presented in Fig. 5. In $\mathcal{F}_{\text{PREP}}$, the ideal adversary specifies all the data that the corrupted parties would like to hold as part of the various sharings generated by the functionality. Namely it specifies the shares, MAC keys and MAC tags. The functionality then completes the sharings while keeping them consistent with the data specified by the adversary.

![Fig. 4. The Ideal Functionality for Computing a Given Function](image)

B An Overview of Online Protocol and Proof of Theorem 1

The online protocol $\Pi_{\text{ONLINE}}$ (presented in Fig. 6) is based on the standard Beaver’s idea of securely evaluating the circuit in a shared fashion using pre-processed shared random multiplication triples $\mathcal{F}$ and shared inputs. Namely, the parties evaluate the circuit $C$ in a $\langle \cdot \rangle$-shared fashion by maintaining the following invariant for each gate in the circuit. Given a $\langle \cdot \rangle$-sharing of the inputs of the gate, the parties generate an $\langle \cdot \rangle$-sharing of the output of the gate. Maintaining the invariant for linear gates requires only local computation, thanks to the linearity property of the $\langle \cdot \rangle$-sharing. For multiplication gates, the parties deploy a shared multiplication triple received from $\mathcal{F}_{\text{PREP}}$ and evaluate the multiplication gate by using Beaver’s trick. Specifically, let $(p, q)$ be the sharing corresponding to the inputs of a multiplication gate and let $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ be the shared random multiplication triple obtained from $\mathcal{F}_{\text{PREP}}$, which is associated with this multiplication gate. To compute an $\langle \cdot \rangle$-sharing of the gate output $p \cdot q$, we note that $p \cdot q = (p - a + a) \cdot (q - b + b) = d \cdot e + d \cdot b + e \cdot a + c$, where $d \overset{\text{def}}{=} p - a$ and $e \overset{\text{def}}{=} q - b$. So if $d$ and $e$ are publicly known then $(p \cdot q) = d \cdot e + d \cdot (b) + e \cdot (a) + (c)$ holds. To make $d$ and $e$ public, the parties first locally compute $\langle d \rangle = \langle p \rangle - \langle a \rangle$ and $\langle e \rangle = \langle q \rangle - \langle b \rangle$ and publicly reconstruct these sharings. Note that even though $d$ and $e$ are made public, the privacy of the gate inputs $p$ and $q$ is preserved, as $a$ and $b$ are random and private. Finally once the parties have the sharing $\langle y \rangle$ for the circuit output, it is publicly reconstructed to enable every party obtain the function output.
The functionality interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ as follows. Let $\mathcal{C} \subset \mathcal{P}$ be the set of corrupted parties.

- **Setup Generation**: On input Setup from the parties in $\mathcal{P}$, the functionality does the following:
  - It creates $n$ public key, secret key pairs $\{\text{pk}^{(i)}, \text{dk}^{(i)}\}_{i=1}^{n}$ of the linearly-homomorphic encryption scheme $\text{HE}$,
  - For each $P_i$, it selects $n$ random values $(r^{(i,1)}, \ldots, r^{(i,n)})$, computes $\text{HE.c}(r^{(i,1)}), \ldots, \text{HE.c}(r^{(i,n)})$ under $\text{pk}^{(i)}$,
  - It sends $(\text{dk}^{(i)}, (r^{(i,1)}, \ldots, r^{(i,n)}), \{\text{pk}^{(i)}\}_{j=1}^{n}, (\text{HE.c}(r^{(i,1)}), \ldots, \text{HE.c}(r^{(i,n)}))\}_{j=1}^{n}$ to party $P_i$.
  - On the behalf of each honest $P_i \in \mathcal{P} \setminus \mathcal{C}$, it sets $\alpha$ and $(\alpha_{ij})_{i,n}$, where the $j$th value is designated to be used in the MAC key for party $P_i$. On the behalf of each corrupted party $P_i \in \mathcal{C}$, it receives from $\mathcal{S}$ the $\alpha_{ij}$ values that $P_i$ wants to use in the MAC keys corresponding to the honest party $P_j$. On receiving, the functionality stores these values.

- **Triple Sharings**: On input Triples from all the parties in $\mathcal{P}$, the functionality generates $\langle \cdot \rangle$-sharing of $\chi$ random multiplication triples in parallel. To generate one such sharing $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$, it does the following:
  - It randomly selects $a$, $b$ and computes $c = ab$. It then runs ‘Single $\langle \cdot \rangle$-sharing Generation’ (see below) for $a$, $b$ and $c$.

- **Input Sharings**: On input $(x^{(i)}, i, \text{Input})$ from party $P_i$ and $(i, \text{Input})$ from the remaining parties, the functionality runs ‘Single $\langle \cdot \rangle$-sharing Generation’ (given below) for $x^{(i)}$.

**Single $\langle \cdot \rangle$-sharing Generation**: The functionality does the following to generate $\langle s \rangle$-sharing for a given value $s$:

- On receiving the shares $\{s_i\}_{P_i \in \mathcal{C}}$ from $\mathcal{S}$ on the behalf of the corrupted parties, it selects a polynomial $S(\cdot)$ of degree at most $t$, such that $S(0) = s$ and $S(i) = s_i$ for each $P_i \in \mathcal{C}$. For $P_i \notin \mathcal{P} \setminus \mathcal{C}$, it computes $s_i = S(i)$.
- On receiving $\{\beta_{ij}\}_{P_i \in \mathcal{C}}$, $\beta_{ij} \notin \mathcal{C}$ from $\mathcal{S}$, the second components of the MAC key that $P_i \in \mathcal{C}$ will have for an honest party $P_j$, it sets $K_{ij} = (\alpha_{ij}, \beta_{ij})$ where $\alpha_{ij}$ was specified by $\mathcal{S}$ in ‘Setup generation’ stage. It computes the MAC tag $\text{MAC}_{K_{ij}}(s_j)$ of $s_j$ for every honest $P_j$ corresponding to the key of every corrupted $P_i$.
- On receiving MAC tags $\{\text{MAC}_{K_{ij}}\}_{P_i \in \mathcal{C}}$, $\beta_{ij} \notin \mathcal{C}$ that the corrupted parties would like to have on their shares $s_i$ corresponding to the MAC key of honest $P_j$, it fixes the key of $P_i$ corresponding to $P_j$ as $K_{ij} = (\alpha_{ij}, \beta_{ij})$ where $\beta_{ij} = \text{MAC}_{K_{ij}} - \alpha_{ij} \cdot s_i$ and $\alpha_{ij}$ was selected by the functionality in ‘Setup generation’ stage.

For every pair of honest parties $(P_j, P_k)$, it chooses the key of $P_j$ as $K_{jk} = (\alpha_{jk}, \beta_{jk})$ where $\alpha_{jk}$ is taken from ‘setup generation phase’ and $\beta_{jk}$ is chosen randomly. It then computes the corresponding MAC tag of $P_j$ as $\text{MAC}_{K_{jk}}(s_k)$.

It sends $\{s_j, \{\text{MAC}_{K_{ij}}, K_{jk}\}_{i,j=1}^{n-1}\}$ to honest party $P_j$ (no need to send anything to corrupted parties as $\mathcal{S}$ has the data of the corrupted parties already).

Fig. 5. Ideal Functionality for Setup Generation, Offline Pre-processing and Input Processing
B.1 Proof of Theorem 1

We refer to \[{}^{10}7{}^{12}12\] for the definition of UC-security. Let \(A\) be a real-world adversary corrupting \(t\) parties during the execution of \(\Pi_{\text{ONLINE}}\) and let \(C \subset P\) denote the set of corrupted parties. We present a simulator \(S_f\) for \(A\), who interacts with \(F_f\) and simulates each received message of \(A\) in the protocol \(\Pi_{\text{ONLINE}}\), from the honest parties and from the functionality \(F_{\text{REP}}\), stage by stage. Note that the simulator will also simulate the steps of the subprotocol \(\text{RecPub}\) executed inside the protocol \(\Pi_{\text{ONLINE}}\). We present the high level idea of the simulator first. The formal details are given in Figure 3.

The idea of \(S_f\) is straightforward. The simulator plays the role of \(F_{\text{REP}}\) and honestly simulates the call to \(F_{\text{REP}}\). Specifically, on receiving Setup from the corrupted parties, it creates \(n\) public key, secret key pairs \(\{pk^{(i)}, dk^{(i)}\}_{1}^{n}\) of the linearly-homomorphic encryption scheme \(\text{HE}\), selects \(n\) random values \((r^{(j,1)}, \ldots, r^{(j,n)}))\) and computes their encryptions \(\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})\) under \(pk^{(i)}\) and sends

\[
\left(\text{dk}^{(i)}, (p^{(i,1)}, \ldots, p^{(i,n)}), \{pk^{(j)}\}_{j=1}^{n}, \{\text{HE}.c(r^{(j,1)}), \ldots, \text{HE}.c(r^{(j,n)})\}_{j=1}^{n}\right)
\]

to every corrupted party \(P_i\). Next, it creates \((\cdot)\)-sharings of \(\chi\) random multiplication triples, taking inputs from \(A\) the shares, the MAC keys and the MAC tags of the corrupted parties in each of these sharings just as in \(F_{\text{REP}}\). Next the simulator learns a corrupted party \(P_i\)'s input \(x^{(i)}\) from \(P_i\)'s input message \((x^{(i)}, i, \text{Input})\) that it sends to \(F_{\text{REP}}\). The simulator sets the input \(x^{(i)}\) of each honest \(P_i\) as zero. With these inputs, \(S_f\) perfectly emulates \(F_{\text{REP}}\). That is, it creates \((\cdot)\)-sharings of the inputs of each party, where \(A\) selects the shares, the MAC keys and the MAC tags of the corrupted parties in each of these sharings just as in \(F_{\text{REP}}\).

Then, the simulator simulates the shared circuit evaluation using the above inputs of the parties, updating the sharings after each gate evaluation. The sharings that need to be opened during the evaluation of multiplication gates are known to the simulator and hence it can easily simulate the messages of the honest parties, corresponding to the protocol steps of \(\text{RecPub}\). On receiving the communication from the corrupted parties (on behalf of the honest parties), it checks if the corrupted parties have sent incorrect information that pass the MAC test or the random combination test whichever is applicable according to \(\text{RecPub}\) protocol steps (for the latter, recall that an honest \(P_j\) is supposed to test if \(c^{(i,j)} = f^{(i,j)}\) for every \(P_i\)). Note that it can do so since it knows all the information that the corrupted parties hold and latter reveal to the honest parties as a part of the sharings that are opened in \(\text{RecPub}\). If it notes that a corrupted party sent wrong information that passes the above tests, then it halts the simulation and aborts. Otherwise, it goes on to simulate the opening of the output sharing as follows:

- The simulator first learns the function output \(y\) by sending the inputs of the corrupted parties to \(F_f\). Since the simulator knows the simulated output sharing, say \(\langle y'\rangle\), it can easily create a “fake” output sharing \(\langle y\rangle\) such that the shares of the corrupted parties in \(\langle y'\rangle\) and \(\langle y\rangle\) are the same.
- The simulator ensures that the shares of the honest parties in the sharing \(\langle y\rangle\) have MAC tags consistent with the MAC keys of the corrupted parties in the sharing \(\langle y'\rangle\). Again the simulator can do this, as it knows the MAC keys of the corrupted parties.
- Having done so, the simulator can now easily send the fake shares of \(y\) and MAC tags on the behalf of the honest parties, which will be consistent with respect to the MAC keys of the parties under \(A\) corresponding to the honest parties. Using these shares, \(A\) reconstructs \(y\), which is the same as in the real protocol. If the corrupted parties sends wrong shares that passes the MAC test, the simulator halts the simulation and aborts.

The differences between the simulated and hybrid views are: (i) First, in simulated view the inputs for the honest parties are not real inputs (0 is used instead) whereas the real inputs of the honest parties are used in the hybrid view. During the simulation of the output opening, the shares (and the MAC tags) of the honest parties corresponding to the output sharing are changed to cook up a sharing corresponding to the actual function output \(y\). This enables the simulator to make sure that the adversary outputs the real \(y\). The adversary/environment cannot make out whether the real inputs or fake (0) inputs are used for the honest parties since the inputs of the honest parties as well as the intermediate values of the circuit computation remain \((\cdot)\)-shared. The public openings during the computation are masked values and therefore they do not leak any information too. So even if the environment has unbounded computing power, it cannot tell apart the views. (ii) Second, in the simulated view, the adversary/environment cannot cheat the simulator.
who is acting on the behalf of the honest parties, by guessing the MAC keys or by guessing the random combiners.

C Realizing $\mathcal{F}_{\text{PREP}}$ in the Synchronous Setting

Here we present a protocol $\Pi_{\text{PREP}}$ which UC-securely realizes the functionality $\mathcal{F}_{\text{PREP}}$ in the synchronous setting. We first discuss the existing primitives and a setup functionality that we require in protocol $\Pi_{\text{PREP}}$.

C.1 Primitives

For $\Pi_{\text{PREP}}$ we assume the following primitives.

**Multi-valued Broadcast Protocol with a Linear Overhead:** Given a public-key set-up, [18] presents a multi-valued broadcast protocol with a Linear Overhead: For $\Pi_C$ Primitives $\Pi_C$ Realizing $F$ and the hybrid views are indistinguishable from the viewpoint of the environment. This completes our proof.

**Threshold Linear Somewhat Homomorphic Encryption (SHE):** In a threshold somewhat homomorphic scheme $\text{SHE}$ with threshold $t$, any party can encrypt a message using the public key of the scheme, while decrypting a ciphertext requires the collaboration of at least $t + 1$ parties. The scheme supports linearly homomorphic operations on ciphertexts; we additionally require it to support one homomorphic multiplication. For practical instantiation of $\text{SHE}$ for our case, we can consider the threshold $\text{SHE}$ schemes of [16,12]. In what follows, we give a high level discussion of the key features of the encryption scheme that we use.

The cryptosystem has a public encryption key $\text{pk}$ and a secret decryption key $\text{dk}_i$ Shamir-shared among the parties with threshold $t$, with each party $P_i$ holding a decryption-key share $\text{dk}_i$. Given $\text{pk}$, a plaintext $x \in \mathbb{F}$ and a randomness $r$, anyone can compute a ciphertext $\text{SHE}.c(x) = \text{SHE}.\text{Enc}_{\text{pk}}(x, r)$, using the encryption algorithm $\text{SHE.\text{Enc}}$, where the size of $\text{SHE}.c(x)$ is $O(\kappa)$ bits. Given a ciphertext $\text{SHE}.c(x) = \text{SHE.\text{Enc}}_{\text{pk}}(x, \ast)$, any party $P_i$ can compute a decryption share $\mu_i = \text{SHE.\text{ShareDec}}_i(\text{dk}_i, \text{SHE}.c(x))$ of $\text{SHE}.c(x)$, consisting of $O(\kappa)$ bits, using its decryption-key share $\text{dk}_i$; here $\text{SHE.\text{ShareDec}}_i$ is the “partial” decryption function for the party $P_i$. Finally, given at least $t + 1$ “correct” decryption shares $\{\mu_i\}$ corresponding to $\text{SHE}.c(x)$, there exists an algorithm which “combines” the decryption shares and outputs the plaintext $x$; since the decryption key is Shamir shared, the combine algorithm simply interpolates a polynomial of degree at most $t$ passing through the given $t + 1$ correct decryption shares and outputs the constant term of the polynomial. The encryption scheme is assumed to be indistinguishable under chosen-plaintext attack (IND-CPA) against a PPT adversary that may know up to $t$ decryption-key shares $\text{dk}_i$. Moreover, given a ciphertext and the corresponding decryption shares of the honest parties, no (additional) information is revealed about the decryption-key shares of the honest parties.

The encryption scheme is linearly homomorphic: given ciphertexts $\text{SHE}.c(x)$ and $\text{SHE}.c(y)$ under the same public key, there exists some operation on ciphertexts, say $\oplus$, such that $\text{SHE}.c(x) \oplus \text{SHE}.c(y) = \text{SHE.\text{Enc}}_{\text{pk}}(x + y)$.

\footnote{We assume an implicit randomness used in a ciphertext until and unless it is explicitly stated.}
Simulator $\mathcal{S}_f$

Let HE be a linearly-homomorphic encryption scheme. The simulator plays the role of the honest parties and simulates each step of the protocol $P_{\text{ONLINE}}$ as follows. The communication of the environment $\mathcal{E}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{E}$ is written on $\mathcal{A}$’s input tape. Likewise, every output value written by $\mathcal{A}$ on its output tape is copied to the simulator’s output tape (to be read by the environment $\mathcal{E}$). The simulator does the following. In the simulation below, we use the following notation: for a $\langle \cdot \rangle$-sharing: (i) the information that corresponds to a party $P_i$ is denoted by $\langle \cdot \rangle_i$ and (ii) the share and (MAC tag, MAC key) of $P_i$ corresponding to $P_j$ is denoted as $\langle \cdot \rangle_{ij}$.

**Simulating the call to $\mathcal{F}_\text{PRep}$:** $\mathcal{S}_f$ honestly emulates $\mathcal{F}_\text{PRep}$ setting the function inputs of the honest parties to be 0. At the end, it knows the following for the triple sharings $\{(a^{(i)})_j, (b^{(i)})_j, (c^{(i)})_j\}_{i=1}^n$: input sharings $\{\langle x^{(i)} \rangle_i\}_{i \in \mathcal{C}}$ of the corrupted parties and the inputs sharings $\{(\tilde{x}^{(i)})_{P_i \in \mathcal{P} \setminus \mathcal{C}}\}$ corresponding to honest (we denote the input of an honest party $P_i$ with $\tilde{x}^{(i)}$ since it is not the real input of $P_i$); (i) $\langle \cdot \rangle_i$ for every honest $P_i$ and (ii) $\langle \cdot \rangle_{ij}$ for every corrupted $P_i \in \mathcal{C}$ and honest party $P_j$. It also knows the combinators $r^{\langle \cdot \rangle}_{i_1, \ldots, r^{\langle \cdot \rangle}_{i_n,n}}$ and the encrypted combinators $\text{HE}.c_{\langle \cdot \rangle_{i_1}}, \ldots, \text{HE}.c_{\langle \cdot \rangle_{i_n,n}}$ for each $P_i \in \mathcal{P}$, along with the public-key secret-key pairs $\{\text{pk}^{(i)}, \text{dk}^{(i)}\}_{i=1}^n$.

**Simulating the Circuit Evaluation:** $\mathcal{S}_f$ simulates the circuit evaluation as follows:

- **Linear Gates:** Since this step involves local computation, $\mathcal{S}_f$ does not have to simulate any messages on the behalf of the honest parties. $\mathcal{S}_f$ locally applies the corresponding linear function on the corresponding gate-input sharings to compute the corresponding gate-output sharing.

- **Multiplication Gates:** These are considered in a batch of size $n(t+1)$. We show the simulation for one batch. Let $\{(a^{(i)})_j, (b^{(i)})_j, (c^{(i)})_j\}_{i=1}^{n(\ell+1)}$ be the pair of sharings corresponding to the input pairs of the $n(t+1)$ multiplication gates which need to be evaluated. Moreover, let $\{(a^{(i)})_j, (b^{(i)})_j, (c^{(i)})_j\}_{i=1}^{n(\ell+1)}$ be the sharing of the corresponding associated multiplication triples. Corresponding to the sharings $\langle d^{(i)} \rangle = (\tilde{a}^{(i)}) - a^{(i)} = (\tilde{b}^{(i)}) - b^{(i)}$ and $\langle e^{(i)} \rangle = (\tilde{e}^{(i)}) - e^{(i)}$, the simulator computes: (i) $\langle \cdot \rangle_j$ for every honest $P_i$ and (ii) $\langle \cdot \rangle_{ij}$ for every corrupted $P_i \in \mathcal{C}$ and honest party $P_j$. It then emulates the messages of the honest parties for the protocol steps of $\text{RecPub}$ to reconstruct $\{\langle d^{(i)} \rangle, e^{(i)}\}_{i=1}^{n(\ell+1)}$ towards the corrupted parties in $\mathcal{C}$. On receiving the messages from the parties in $\mathcal{C}$ in $\text{RecPub}$ on behalf of the honest parties, if $\mathcal{S}_f$ finds them incorrect but still passing either the MAC test or the linear combiner test, then it halts simulation and aborts. Finally, corresponding to the following sharings, $\langle \tilde{y}^{(i)} \cdot \tilde{y}^{(i)} \rangle = d^{(i)} e^{(i)} + d^{(i)} (b^{(i)}) + e^{(i)} (a^{(i)}) + c^{(i)}$, the simulator computes: (i) $\langle \cdot \rangle_j$ for every honest $P_i$ and (ii) $\langle \cdot \rangle_{ij}$ for every corrupted $P_i \in \mathcal{C}$ and honest party $P_j$.

- **Output Gate:** Corresponding to each $P_i \in \mathcal{C}$, the simulator calls $\mathcal{F}_f$ with $\langle \cdot \rangle$, and obtains the function output $y$. Let $\langle \tilde{y} \rangle$ be the sharing associated with the output gate, $\mathcal{S}_f$ at this stage knows $\langle \tilde{y} \rangle$, for each honest $P_i$ and $\langle \tilde{y} \rangle_{ij}$ of each corrupted $P_i$ and honest $P_j$. The simulator then simulates the messages of the honest parties corresponding to the instances of $\text{RecPrv}(\langle \tilde{y} \rangle, \star)$ as follows:

  - Let $\{\tilde{y}_j\}_{j \in \mathcal{C}}$ be the shares of the corrupted parties corresponding to $\langle \tilde{y} \rangle$. Using these shares and the value $y$, the simulator generates the simulated output sharing $\langle y \rangle$, such that the shares of the corrupted parties $P_i \in \mathcal{C}$ are same in $\langle y \rangle$ and $\langle \tilde{y} \rangle$.
  - For every corrupted $P_i \in \mathcal{C}$ and every honest $P_j \in \mathcal{P} \setminus \mathcal{C}$, $\mathcal{S}_f$ does the following. It knows $K_{\text{MAC}}^{(y)}$, the MAC key of $P_i$, corresponding to each honest $P_j$ in the sharing $\langle \tilde{y} \rangle$. Moreover, let $y_j$ be the share of $y$ for $P_i$ in the sharing $\langle \tilde{y} \rangle$. It then computes the simulated MAC tag $\text{MAC}_{\text{MAC}}^{(y_j)}(y_j)$ for the share $y_j$ under the MAC key $K_{\text{MAC}}^{(y)}$.
  - On the behalf of each honest $P_j \in \mathcal{P} \setminus \mathcal{C}$, the simulator sends the share $y_j$ and the MAC tags $\text{MAC}_{\text{MAC}}^{(y_j)}(y_j)$ to $\mathcal{A}$ as part of $\text{RecPrv}(\langle \tilde{y} \rangle, \star)$, for each $P_i \in \mathcal{C}$.
  - It receives the shares and MAC tags from the corrupted parties on the behalf of the honest parties. If it sees that the corrupted parties sent wrong information that pass the MAC test, then it aborts.

Finally $\mathcal{S}_f$ outputs $\mathcal{A}$’s output.

Fig. 6. Simulator for the adversary $\mathcal{A}$ corrupting $t$ parties in the set $\mathcal{C} \subset \mathcal{P}$ during the protocol $P_{\text{ONLINE}}$. 

16
Moreover, given a ciphertext $\text{SHE}.c(x)$ and a publicly known constant $c$, using the linearly homomorphic property, one can compute $\text{SHE}.c(c \cdot x)$. One can homomorphically compute $\text{SHE}.c(x - y)$, given encryptions $\text{SHE}.c(x)$ and $\text{SHE}.c(y)$ under the same public key. Finally, the cryptosystem is one-time multiplicatively homomorphic. That is, given encryptions $\text{SHE}.c(x)$ and $\text{SHE}.c(y)$ (under the same public key), there exists some operation on ciphertexts, say $\Box$, such that $\text{SHE}.c(x) \Box \text{SHE}.c(y) = \text{SHE}.\text{Enc}_{pk}(x \cdot y)$. The operation $\Box$ can be applied at most once; a ciphertext that is obtained after applying $\Box$ more than once may not guarantee correct decryption.

**UC-secure Zero-knowledge (ZK) Proofs:** We assume to have efficient UC-secure ZK protocols for the following properties:

- **Proof of Plain-text (PoPK) for SHE:** In this protocol, there exists a prover $P_i \in \mathcal{P}$, who computes encryptions of $\ell$ values under the public key $pk$ of SHE and sends the encryptions to every party in $\mathcal{P}$; using this protocol, $P_i$ can prove to everyone the knowledge of underlying plaintexts. We assume that the protocol has communication complexity $O(\kappa \cdot n \cdot \ell)$ bits. This is achievable by combining the multi-valued broadcast protocol of [13] (provided $\ell$ is sufficiently large) and any 2-party non-interactive ZK (NIZK) protocol for PoPK for SHE (see for example [16,11,12]). The idea is the following: assume that the communication complexity of the 2-party NIZK protocol for PoPK is $O(\kappa)$ bits; then $P_i$ broadcasts the $\ell$ ciphertexts, along with the NIZK proof for each ciphertext (so total $O(\kappa \cdot \ell)$ bits) via the multi-valued broadcast protocol of [13]. Assuming $\ell = \Theta(n^3 \cdot (n + \kappa))$, the total communication complexity becomes $O(\kappa \cdot n \cdot \ell)$ bits.

- **Proof of Correct Decryption (PoCD) for SHE:** In this protocol, there exists a designated prover and a verifier, along with a publicly known ciphertext under the public key $pk$ of SHE. The prover computes a decryption share of the ciphertext using its decryption-key share and sends the decryption share to the verifier. Using this protocol, the prover proves that it has correctly computed the decryption share. We assume that the protocol has communication complexity $O(\kappa)$ bits (for example, see [16,11,12]).

- **Proof of Plain-text (PoPK) for HE:** In this protocol, there exists a prover $P_i \in \mathcal{P}$ with public key $pk^{(i)}$ who computes encryptions of $\ell$ values under $pk^{(i)}$ and sends the encryptions to a verifier $P_j \in \mathcal{P}$. Using this protocol, $P_i$ can prove to $P_j$ the knowledge of underlying plaintexts. We assume that the protocol has communication complexity $O(\kappa)$ bits.

- **Proof of Correct Multiplication (PoCM) for HE:** In this protocol, there exists a prover $P_i \in \mathcal{P}$ and a verifier $P_j \in \mathcal{P}$, with $P_i$ and $P_j$ knowing values $\alpha$ and $\beta$ respectively. Moreover an encryption $\text{HE}.c(\alpha)$ of $\alpha$ under the public key $pk^{(i)}$ and an encryption $\text{HE}.c(\alpha)$ of $\alpha$ under the public key $pk^{(j)}$ of HE are known to $P_i$ and $P_j$. Prover $P_i$ selects some $\beta$ and homomorphically computes an encryption $\text{HE}.c(\gamma) = \alpha \oplus \text{HE}.c(\alpha) \oplus \text{HE}.\text{Enc}_{pk^{(j)}}(\beta, \star)$ of $\gamma = \alpha \cdot \alpha + \beta$ and sends $\text{HE}.c(\gamma)$ to $P_j$; note that $\text{HE}.c(\gamma)$ will be under the key $pk^{(i)}$. Using this protocol, prover $P_j$ can prove to $P_i$ that it has computed $\text{HE}.c(\gamma)$ as above. The protocol has communication complexity $O(\kappa)$ bits (see [7] for example).

**Efficient Distributed Decryption with a Fewer Number of ZK Proofs:** In protocol $\Pi_{\text{REP}}$, we will require to decrypt several ciphertexts encrypted under SHE. One obvious way of doing the distributed decryption is to ask each party to compute a decryption share of the ciphertext and send the same to the designated party, along with a proof of correct decryption; this will unfortunately require $O(n^2)$ instances of PoCD for publicly decrypting a single ciphertext and hence $O(n^2 \cdot \ell)$ such instances will be required to decrypt $\ell$ ciphertexts. Instead we borrow a protocol DistDec from [12], which overall requires $O(n^3)$ instances of PoCD to decrypt $\ell$ ciphertexts. The idea is to exchange the decryption shares without any PoCD and then use the error-detection to detect if the decryption shares are correct. This is always possible as the decryption key of the SHE is Shamir-shared with threshold $\ell < n/2$ and so the error-detection properties of the Reed-Solomon (RS) codes are applicable. In case any error is detected, then a PoCD proof is demanded from every party. Obviously the honest decryption-share providers will successfully give the PoCD proof, while a malicious decryption-share provider will fail to do so. The corresponding decryption shares are then ignored and using the remaining decryption shares, the ciphertext can be decrypted back correctly. Once a malicious decryption-share provider is identified, it is ignored in all the future instances of the distributed decryption. As there exist at least $t+1$ honest parties whose decryption shares will be correctly identified by every honest party, the process will always terminate correctly for every honest party.
While executing DistDec, a corrupted party may un-necessarily demand a PoCD even if no error is detected at its end. To prevent him from always doing the same, every party maintains a local counter to count the number of times a PoCD request is received from a specific receiver; if the counter exceeds the value $t$ then definitely the receiver is corrupted. This is because for an honest receiver, error will be detected for at most $t$ instances after which it will know all the $t$ corrupted parties. So if at all a receiver requests PoCD more than $t$ times then definitely the receiver is corrupted. We note that during the protocol $\Pi_{\text{PREP}}$, we will use distributed decryption to decrypt certain ciphertexts only towards some designated parties, as well as for publicly decrypting certain ciphertexts. However, irrespective of the case, DistDec ensures that the total number of PoCD is $O(n^3)$, which is independent of the circuit size; for details see [12].

C.2 The Ideal One-time Setup Functionality

For $\Pi_{\text{PREP}}$, we assume an ideal set-up functionality $F_{\text{ONE-TIME-SETUP}}$, presented in Figure 7. The functionality creates the following one-time set-up for the $n$ parties:

- A public key, secret key pair for a threshold SHE scheme with threshold $t$ is generated and each party is given the public key and its decryption-key share.
- A public key, secret key pair for the linearly-homomorphic encryption scheme is generated for every party. The secret key is give to the corresponding party while the public key is given to all the parties.
- On the behalf of each party, $n$ random values are selected and encrypted under its public key of the HE scheme. The random values are sent to that party while its encryptions are given to all the parties.
- On the behalf of each honest party, $n$ random $\alpha$ values are selected and given to it. The $\alpha$ values are designated to be used by that honest party as the $\alpha$-component of the MAC key for different parties across all sharings to be used for a computation. The functionality creates encryptions of these $\alpha$ values and each encryption is sent to the respective parties. In addition, all the encryptions and their corresponding randomness are sent to the honest party.
- From every corrupted party, the functionality receives $\alpha$ values which it wants to use as the $\alpha$-component of the MAC keys corresponding to the honest parties. In addition, the functionality also receives the encryptions of these $\alpha$ values under the public key of the HE scheme of the corrupted party. If the encryptions are valid, then the functionality stores these $\alpha$ values and sends their encryptions to the respective honest parties.

C.3 Protocol $\Pi_{\text{PREP}}$

Protocol $\Pi_{\text{PREP}}$ is presented in Figure 8. We present a high level overview of the protocol. The parties first call the functionality $F_{\text{ONE-TIME-SETUP}}$ and generate the required setup. Next the parties generate $\langle \cdot \rangle$-sharing of “large” number of random and private key-consistent $\langle \cdot \rangle$-shared multiplication triples, say $\chi$ triples, where $\chi \geq M + n$. We explain how one such random sharing $\langle (a), (b), (c) \rangle$ is generated; in the protocol the same steps are executed in parallel for $\chi$ batches and the broadcasts required for all the $\chi$ batches are done in parallel via the multi-valued broadcast protocol of [18].

The generation of $\langle (a), (b), (c) \rangle$ is done in two stages: in the first stage, the parties first generate $\langle [a], [b], [c] \rangle$ and in the second stage, the parties generate pair-wise MACs to transform $\langle [a], [b], [c] \rangle$ to $\langle (a), (b), (c) \rangle$. To generate $\langle [a], [b], [c] \rangle$, we use an idea similar to [10], extended for Shamir sharing. Specifically, to generate $\langle [a] \rangle$, we ask $t + 1$ parties (say the first $t + 1$ parties) to define a polynomial of degree at most $t$ in a shared fashion, where each party contributes one random point on the polynomial. For this, each party $P_i \in \{P_1, \ldots, P_{t+1}\}$ selects a random $a_i$, encrypts it using the public key of the SHE scheme and broadcast the ciphertext to every party, with a ZK proof of the underlying plaintext. Next, we define the polynomial $A(\cdot)$ of degree at most $t$, passing through the points $(1, a_1), \ldots, (t + 1, a_{t+1})$. Since there exists at least one honest party in the set $\{P_1, \ldots, P_{t+1}\}$ whose corresponding $a_i$ value will be random and private, it follows that we will have one degree of freedom in the $A(\cdot)$ polynomial. So if we define $a_i \stackrel{\text{def}}{=} A(i)$ for every $P_i \in \{P_{t+2}, \ldots, P_n\}$, then clearly the vector $(a_1, \ldots, a_n)$ defines a sharing of $[a]$ of the value $a \stackrel{\text{def}}{=} A(0)$. The $t + 1$ parties in $\{P_1, \ldots, P_{t+1}\}$ will already have their shares corresponding to $[a]$. So to complete $[a]$, all we need to do is to ensure that the remaining parties $P_i \in \{P_{t+2}, \ldots, P_n\}$ also obtain their shares $A(i)$. For this we use the fact that the values $\{A(i)\}_{i=t+2}$ are publicly known linear combinations of the $t + 1$ values.
The functionality interacts with the set of parties $\mathcal{P}$ and the adversary $\mathcal{S}$. Let $\mathcal{C} \subset \mathcal{P}$ be the set of corrupted parties, with $|\mathcal{C}| \leq t$. Upon receiving (init) from all the parties, the functionality does the following:

- Creating Threshold SHE Setup: the functionality computes a public-key, secret-key pair $(pk, dk)$ of the threshold SHE scheme with threshold $t$, along with the secret-key shares $dk_1, \ldots, dk_\alpha$ for the $\alpha$ parties. To every party $P_i \in \mathcal{P}$ it then sends $(pk, dk_i)$.

- Creating Public/Secret keys of HE for Every Party: It creates $n$ public key, secret key pairs $\{pk^{(i)}, dk^{(i)}\}_{i=1}^n$ of the linearly-homomorphic encryption scheme HE and sends $dk^{(i)}$ to party $P_i$ for $i = 1, \ldots, n$ and sends $\{pk^{(j)}\}_{j=1}^n$ to all the parties.

- Creating Encrypted Combiners on the Behalf of Each Party: For every $P_j \in \mathcal{P}$, it selects $n$ random values $R_{ij} = \{r^{(j,1)}, \ldots, r^{(j,n)}\}$ and computes the encryptions $HE.c(r^{(j,i)}) = HE.Enc_{pk^{(j)}}(r^{(j,i)} \cdot \star)$ for $i = 1, \ldots, n$. It then sends $R_{ij}$ to the party $P_j$ and the encryptions $HE.c(r^{(j,1)}), \ldots, HE.c(r^{(j,n)})$ to all the parties.

- Creating Encrypted MAC Keys on the Behalf of Honest Parties: For every honest party $P_i \in \mathcal{P} \setminus \mathcal{C}$, the functionality selects $n$ random values $\alpha_1, \ldots, \alpha_\alpha$ and computes the encryptions $HE.c(\alpha_j) = HE.Enc_{pk^{(j)}}(\alpha_j, \star)$ for $j = 1, \ldots, n$. The functionality then sends $\{\alpha_{ij}, HE.c(\alpha_{ij})\}_{i=1}^\alpha$ to $P_i$, along with the randomness used in the encryptions. To every party $P_j \in \mathcal{P}$, the functionality sends $HE.c(\alpha_{ij})$.

- Sending Encrypted MAC Keys on the Behalf of Corrupted Parties Corresponding to Honest Parties: On the behalf of every corrupted party $P_i \in C$, the functionality receives from $S$ the $\alpha_{ij}$ values that $P_i$ would like to use as the $\alpha$-component of the MAC key for each honest $P_j \in \mathcal{P} \setminus C$. For each such $\alpha_{ij}$ value, the functionality also receives from $S$ an encryption $HE.c(\alpha_{ij})$ of $\alpha_{ij}$ under the public key $pk^{(i)}$. The functionality verifies if $HE.Dec_{dk^{(i)}}(HE.c(\alpha_{ij})) \equiv \alpha_{ij}$. If the verification passes then it stores $\alpha_{ij}$ on the behalf of $P_i$ and sends the encryption $HE.c(\alpha_{ij})$ to the party $P_j$.

Now using similar procedure as above, the parties compute $[b]$ and $[r]$, for a random $b$ and $r$, along with an encryption of $b$ and $r$. To compute $[c]$ from $[a]$ and $[b]$, we use the multiplicative homomorphic property of the SHE. Specifically from encryptions of $a, b$, the parties homomorphically compute an encryption of $c = a \cdot r$.

To generate the pair-wise MACs on the shares of $a$, $b$, and $c$ is straightforward and is done using the same procedure as BDOZ; we explain how this is done for the shares of $a$. Consider the pair of parties $(P_i, P_j)$, with party $P_i$ holding the share $a_i$, on which it wants to compute the MAC tag under the MAC key $K_{ij} = (\alpha_{ij}, \beta_{ij})$, held by party $P_j$. Note that if $P_i$ is honest then any encryption $HE.c(\alpha_{ij})$ of $\alpha_{ij}$ under $pk^{(j)}$ will be already known to $P_i$ from $\mathcal{F}_\text{ONE-TIME-SETUP}$. To compute the MAC tag, party $P_i$ encrypts $a_i$ under its public key $pk^{(i)}$ and sends the encryption to $P_j$, along with a ZK PoPK. Party $P_j$ then homomorphically computes an encryption of $MAC_{\{a_i\}}(a_i)$ under $pk^{(i)}$ and sends the same to $P_i$, along with a ZK PoCM. Party $P_i$ then obtains the tag after decrypting the encrypted tag.

If $P_i$ is honest, then $P_j$ learns nothing about $a_i$, thanks to the semantic security of the linearly-homomorphic encryption scheme. By following the above procedure, it is ensured that every pair of honest parties have consistent MAC tags and keys. To ensure that a corrupted $P_i$ uses consistent MAC keys for an honest $P_j$ across various sharings, PoCM is used. It is interesting to note that during the pair-wise MAC generation, we do not check whether $P_i$ is sending encryptions of the correct shares to the parties; if $P_i$ does not do the same, it ends up getting incorrect MAC tags with respect to the MAC keys of honest $P_j$ and so later its share will be discarded by the honest $P_j$ during the reconstruction protocol.

We note that the pair-wise MACs could be setup even by using the threshold SHE; however this will make the overhead of the protocol $\mathcal{O}(n^3)$ per multiplication gate, instead of $\mathcal{O}(n^2)$; this is because there will be $\mathcal{O}(n^2)$ pairwise encrypted MAC tags for each sharing and decrypting one ciphertext via distributed decryption involves $\mathcal{O}(n)$ overhead.

---

**Fig. 7.** Ideal Functionality for One-time Setup
In the protocol, each party needs to broadcast $O(\chi)$ ciphertexts and NIZK proofs. If $\chi$ is sufficiently large then using the broadcast protocol of \cite{18}, this will cost in total $O(\kappa \cdot (n^2 \cdot \chi))$ bits. There will be $O(\chi)$ ciphertexts which need to be publicly decrypted, costing $O(\kappa \cdot (n^2 \cdot \chi))$ bits of communication. In addition, $O(n \cdot \chi)$ ciphertexts need to be decrypted towards designated parties, costing $O(\kappa \cdot (n^3 \cdot \chi))$ bits of communication. The decryptions (both public and private) are done via the protocol DistDec and in total $O(n^3)$ instances of PoCD may be required, costing $O(\kappa \cdot n^3)$ bits of communication. Finally setting up the pair-wise MACs will cost $O(\kappa \cdot (n^2 \cdot \chi))$ bits of communication. So the communication complexity of $\Pi_{\text{prep}}$ will be $O(\kappa \cdot (n^2 \cdot \chi + n^3))$ bits.

\section{Securely Realizing $F_{\text{prep-abort}}$ with $t < n/2$ in the Partial Asynchronous Setting}

In \cite{14} it was shown how to securely realize a variant of $F_{\text{prep}}$, augmented with abort (we call this functionality $F_{\text{prep-abort}}$), in the partial asynchronous setting with $t < n/3$. The functionality $F_{\text{prep-abort}}$ is similar to $F_{\text{prep}}$ except that the functionality distributes the generated values to the honest parties depending upon the choice of the adversary: if adversary sends an OK signal then $F_{\text{prep-abort}}$ distributes the information to the honest parties in the same way as done in $F_{\text{prep}}$, otherwise it sends ⊥ to the honest parties.

To securely realize $F_{\text{prep-abort}}$ in the partial asynchronous setting with $t < n/3$, \cite{14} used the following approach: assume there exists a secure realization of $F_{\text{prep}}$, say $\Pi_{\text{prep}}$, in a synchronous setting with broadcast (in \cite{14}, two instantiations of $\Pi_{\text{prep}}$ are presented). Protocol $\Pi_{\text{prep}}$ is then executed in the partial asynchronous setting in a “special” way to ensure that no additional information is revealed prematurely (more on this in the sequel). Finally the parties run an ABA protocol to agree on whether the preprocessing succeeded and accordingly they either abort or successfully terminate the protocol. From the above discussion, it is clear that we can securely realize $F_{\text{prep-abort}}$ with $t < n/2$ in the partial asynchronous setting following the blueprint of \cite{14}, provided that we have the following two components:

\begin{itemize}
  \item \textbf{(a)} A secure realization of $F_{\text{prep}}$ with $t < n/2$ in the synchronous communication setting with broadcast;
  \item \textbf{(b)} An ABA protocol with $t < n/2$. The former is presented in the previous section. For the latter, we consider the following two options (recall that ABA protocol with $t < n/2$ is impossible to achieve):
\end{itemize}

\begin{itemize}
  \item \textbf{Synchronous Communication Rounds after the Synchronization Point:} It is well known that $t + 1$ synchronous communication rounds are sufficient to achieve agreement with $t < n/2$ in the computational setting \cite{13}. So if we assume $t + 1$ synchronous communication rounds after the synchronization point, then we can achieve agreement among the $n$ parties on whether $\Pi_{\text{prep}}$ when executed in the partial asynchronous setting, succeeded or not. Alternatively, one can use a constant number of synchronous communication rounds after the synchronization point and run constant expected round synchronous agreement protocols \cite{23}.
  \item \textbf{Non-equivocation Mechanism:} In \cite{13} it is shown how to design agreement protocols with $t < n/2$ in an asynchronous setting, provided there is a mechanism to enforce “non-equivocation”. On a very high level, such a mechanism prevents a corrupted party to transmit conflicting messages to honest parties; however a corrupted party may send messages to certain number of parties and decide not to communicate to the rest of the parties. So such a mechanism is strictly weaker than the broadcast primitive. In \cite{13} it is also discussed how such a non-equivocation mechanism can be securely realized assuming a trusted hardware module with each party. One can use such a non-equivocation mechanism to agree about the status of $\Pi_{\text{prep}}$.
\end{itemize}

Now we discuss how to run the protocol $\Pi_{\text{prep}}$ in the partial asynchronous setting following the method of \cite{14}. The basic idea is to execute $\Pi_{\text{prep}}$ over an asynchronous network where a (honest) party $P_i$ starts computation for round $i + 1$ only when it receives all the communication that it is supposed to receive in the $i$th round (for instance, if the $i$th step specifies that it should receive some information from all the parties, then it waits to receive some information from all the parties and not just from $n-t$ parties). The messages which are supposed to be communicated over the point-to-point channels are sent to the designated receivers. Any message which are supposed to be broadcast (such a message is denoted as broadcast message) by $P_i$, is sent by $P_i$ to all the $n$ parties via point-to-point channels. Thus each instance of broadcast is replaced (simulated) by $n$ communication over point-to-point channels. A corrupted sender may not simulate the broadcast properly. So once all the communication rounds of $\Pi_{\text{prep}}$ are executed, the parties exchange among themselves all the broadcast messages they received from different parties in different rounds. Finally every party $P_i$ sets its status bit $q_i$ for $\Pi_{\text{prep}}$ to 1 if \textbf{all} the following conditions hold. Otherwise it sets $q_i = 0$:
Protocol Π_{PREP}

One-time Setup Generation: Each $P_i \in \mathcal{P}$ calls $\mathcal{F}_{\text{ONETIME-SETUP}}$ with (init) and obtains $\text{pk}, \text{dk}_{\text{key}}, \{\text{pk}^{(j)}\}_{j=1}^n, \text{dk}^{(i)}, R^{(i)} = (r^{(i,1)}, \ldots, r^{(i,1)})$ and $(\{\text{HE}.c(r^{(j,n)})\})_{j=1}^n$. In addition, the parties also receive following information:

- If $P_i$ is honest then it receives $\{\alpha_{ij}\}_{j=1}^n$, encryptions $\text{HE}.c(\alpha_{ij})$ under $\text{pk}^{(i)}$ and the randomness used in these encryptions; moreover, for every $P_j \in \mathcal{P}$, party $P_i$ also receives the encryptions $\text{HE}.c(\alpha_{ji})$ under $\text{pk}^{(j)}$.
- If $P_i$ is corrupted, then corresponding to every honest $P_i$, it receives the encryption $\text{HE}.c(\alpha_{ji})$ under $\text{pk}^{(j)}$.

Generating Shared Multiplication Triples: The parties generate in parallel $(\cdot \cdot \cdot)$-sharing of $\chi$ random multiplication triples, where $\chi \geq M + n$. The following steps are executed to generate one such shared multiplication triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$:

- Generating the Sharings $\langle [a], [b], [c], [r] \rangle$ — the parties do the following:
  - Every party $P_i \in \{P_1, \ldots, P_{t+1}\}$ selects three random values $(a_i, b_i, r_i)$, computes the encryptions $\text{SHE}.c(a_i) = \text{SHE}.\text{Enc}_{\text{pk}}(a_i, \ast), \text{SHE}.c(b_i) = \text{SHE}.\text{Enc}_{\text{pk}}(b_i, \ast)$ and $\text{SHE}.c(r_i) = \text{SHE}.\text{Enc}_{\text{pk}}(r_i, \ast)$. It then broadcasts these encryptions, along with a 2K PoPK; the encryptions and the corresponding proofs for all the $\chi$ batches are broadcast together.
  - Let the set of $t+1$ values $\{(a_i, a_i)\}_{i=1}^{t+1}, \{(b_i, b_i)\}_{i=1}^{t+1}$ and $\{(i, r_i)\}_{i=1}^{t+1}$ define polynomials $A(\cdot), B(\cdot)$ and $R(\cdot)$ respectively. Define $a \overset{\text{def}}{=} A(0), b \overset{\text{def}}{=} B(0)$ and $r \overset{\text{def}}{=} R(0)$.
  - From the $t+1$ encryptions $\{\text{SHE}.c(a_i)\}_{i=1}^{t+1}$, the parties homomorphically compute the encryptions $\text{SHE}.c(a)$ and encryptions $\text{SHE}.c(A(t+2)), \ldots, \text{SHE}.c(A(n))$. Similarly, the parties homomorphically compute the encryptions $\text{SHE}.c(b)$ and encryptions $\text{SHE}.c(B(t+2)), \ldots, \text{SHE}.c(B(n))$ from the $t+1$ encryptions $\{\text{SHE}.c(b_i)\}_{i=1}^{t+1}$. In the same way, the parties homomorphically compute the encryptions $\text{SHE}.c(r)$ and encryptions $\text{SHE}.c(R(t+2)), \ldots, \text{SHE}.c(R(n))$ from the $t+1$ encryptions $\{\text{SHE}.c(r_i)\}_{i=1}^{t+1}$.
  - The parties homomorphically compute $\text{SHE}.c(e) = \text{SHE}.c(\alpha) \boxplus \text{SHE}.c(b)$, followed by homomorphically computing $\text{SHE}.c(e + r) = \text{SHE}.c(e) \boxplus \text{SHE}.c(r)$.
  - The encryptions $\text{SHE}.c(\langle A(t+2) \rangle), \text{SHE}.c(\langle B(t+2) \rangle), \ldots, (\text{SHE}.c(\langle A(n) \rangle), \text{SHE}.c(\langle B(n) \rangle), \text{SHE}.c(\langle R(n) \rangle)$ are decrypted (in a distributed fashion) towards party $P_{t+2}, \ldots, P_n$ respectively. In addition, the encryption $\text{SHE}.c(e + r)$ is distributed decrypted publicly. For doing these distributed decryptions, the parties execute the protocol DistDec.
  - Parties $P_{t+2}, \ldots, P_n$ sets $(a_{t+2} = A(t+2), b_{t+2} = B(t+2), r_{t+2} = R(t+2), \ldots, (a_n = A(n), b_n = B(n), r_n = R(n))$ respectively. This completes the generation of $[a], [b], [c]$.
  - The parties compute $[e] = [c] + [r] - [r]$.

- Generating Pair-wise MACs — the parties generate in parallel pair-wise MACs on the $[\cdot \cdot \cdot]$-shared multiplication triples. The following steps are executed for getting the MACs on one such sharing $[a]$:
  - Every party $P_i$ computes an encryption $\text{HE}.c(a_i) = \text{HE}.\text{Enc}_{\text{pk}^{(i)}}(a_i, \ast)$ of its shares $a_i$ and broadcasts the same, along with a 2K PoPK.
  - Every pair of parties $(P_i, P_j)$ execute the following steps:
    - Party $P_i$ selects a random $\beta_{ji}$ to form the MAC key $K_{ji} \overset{\text{def}}{=} (\alpha_{ji}, \beta_{ji})$.
    - Party $P_i$ homomorphically computes $\text{HE}.c(\langle \text{MAC}_{K_{ji}}(a_i) \rangle) \overset{\text{def}}{=} \alpha_{ji} \oplus \text{HE}.c(a_i) \oplus \text{HE}.\text{Enc}_{\text{pk}^{(i)}}(\beta_{ji}, \ast)$ and sends $\text{HE}.c(\langle \text{MAC}_{K_{ji}}(a_i) \rangle)$ to $P_j$. Party $P_i$ and $P_j$ then executes an instance of PoCM, with $P_i$ and $P_j$ playing the role of the prover and verifier respectively. If PoCM is successful, then $P_i$ computes $\text{MAC}_{K_{ji}}(a_i) = \text{HE}.\text{Dec}_{\text{dk}^{(j)}}(\text{HE}.c(\langle \text{MAC}_{K_{ji}}(a_i) \rangle))$.

Input Stage: Let party $P_i \in \mathcal{P}$ has the input $x^{(i)}$ for the computation. The parties associate a sharing from the shared triples, say $\langle a^{(i)} \rangle$, as the shared mask for $P_i$ and do the following to generate $\langle x^{(i)} \rangle$:

- Execute $\text{RecPrv}(P_i, \langle a^{(i)} \rangle)$ to enable $P_i$ robustly reconstruct $a^{(i)}$.
- Party $P_i$ then broadcasts the masked input $m^{(i)} = a^{(i)} + x^{(i)}$.
- The parties set $\langle x^{(i)} \rangle = m^{(i)} - \langle a^{(i)} \rangle$.

---

*a* If $P_j$ is honest, then $\alpha_{ji}$ will be available to $P_j$ from $\mathcal{F}_{\text{ONETIME-SETUP}}$ and encryption $\text{HE}.c(\alpha_{ji})$ under $\text{pk}^{(j)}$ will be available to $P_j$ and $P_i$ from $\mathcal{F}_{\text{ONETIME-SETUP}}$, with $P_i$ also knowing the randomness for the encryption. If $P_j$ is corrupted and $P_i$ is honest, then an encryption $\text{HE}.c(\alpha_{ji})$ under $\text{pk}^{(j)}$ will be available to $P_i$ from $\mathcal{F}_{\text{ONETIME-SETUP}}$.

---

Fig. 8. Protocol for Realizing $\Pi_{PREP}$ in the $\mathcal{F}_{\text{ONETIME-SETUP}}$-hybrid Model in the Synchronous Setting
- $P_i$ received all the messages it is supposed to receive in $\Pi_{\text{PREP}}$ before the timeout. In addition, $P_i$ sent all the message that it is supposed to send in $\Pi_{\text{PREP}}$ before timeout.
- $P_i$’s received broadcast messages in $\Pi_{\text{PREP}}$ are the same as those received by all other parties.
- No instance of distributed decryption during the instances of DistDec in $\Pi_{\text{PREP}}$ fails for the party $P_i$. Recall that in DistDec if $P_i$ detects any error in the distributed decryption, then it demands for a PoCD. We no longer need to execute the PoCD step in the instances of DistDec in the partial asynchronous setting. Because if $P_i$ detects any error then it stops participating in further rounds of $\Pi_{\text{PREP}}$. This ensures that every other honest party will stop participating in $\Pi_{\text{PREP}}$ from the next round onwards.

After the timeout, the parties execute an instance of BA with input $q_i$. If the output of the BA protocol is 1, then the execution of $\Pi_{\text{PREP}}$ is successful and the parties proceed to execute the protocol $\Pi_{\text{ONLINE}}$ with the values generated at the end of $\Pi_{\text{PREP}}$; otherwise the parties abort the protocol. The BA protocol ensures that if $q_i = 1$ for all the honest parties, then $\Pi_{\text{PREP}}$ is successful and if $q_i = 0$ for all the honest parties then parties will abort $\Pi_{\text{PREP}}$. If the output of the BA protocol is 1 then it implies that at least one honest party $P_i$ has input $q_i = 1$ for the BA protocol and so $\Pi_{\text{PREP}}$ was terminated successfully for $P_i$ before the timeout. This further implies that the local view of each honest party contributed by the set of honest parties are consistent with each other till the timeout.

It is easy to see that $\Pi_{\text{PREP}}$ when executed in the partial asynchronous setting will have communication complexity $O(\kappa \cdot (n^2 \cdot \chi + \text{poly}(n)))$ bits. Moreover the protocol securely realizes $\mathcal{F}_{\text{PREP-ABORT}}$. 