Linear Overhead Optimally-resilient Robust MPC Using Preprocessing

Ashish Choudhury[1], Emmanuela Orsini[2], Arpita Patra[3], and Nigel. P. Smart[2]

[1] International Institute of Information Technology Bangalore, India.
ashish.choudhury@iiitb.ac.in
[2] Dept. Computer Science, University of Bristol, Bristol, United Kingdom.
Emmanuela.Orsini@bristol.ac.uk,nigel@cs.bris.ac.uk
[3] Department of Computer Science and Automation, Indian Institute of Science, India.
arpita@csa.iisc.ernet.in

Abstract. We present a new technique for robust secret reconstruction with $\mathcal{O}(n)$ communication complexity. By applying this technique, we achieve $\mathcal{O}(n)$ communication complexity per multiplication for a wide class of robust practical Multi-Party Computation (MPC) protocols. In particular our technique applies to robust threshold computationally secure protocols in the case of $t < n/2$ in the pre-processing model. Previously in the pre-processing model, $\mathcal{O}(n)$ communication complexity per multiplication was only known in the case of computationally secure non-robust protocols in the dishonest majority setting (i.e. with $t < n$) and in the case of perfectly-secure robust protocols with $t < n/3$. A similar protocol was sketched by Damgård and Nielsen, but no details were given to enable an estimate of the communication complexity. Surprisingly our robust reconstruction protocol applies for both the synchronous and asynchronous settings.

1 Introduction

Secure MPC is a fundamental problem in secure distributed computing [33,27,8,14]. An MPC protocol allows a set of n mutually distrusting parties with private inputs to securely compute a joint function of their inputs, even if t out of the n parties are corrupted. Determining the communication complexity of MPC in terms of n, 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 $\mathcal{O}(n)$ per multiplication in the computation. However, "most" *robust* secret-sharing based MPC protocols which are practical have complexity $\mathcal{O}(n^2)$.

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-theoretic passively secure sub-protocol for multiplication gates when $t < n/2$ (locally multiply the shares, reshare and then recombine) we require $\mathcal{O}(n^2)$ messages per multiplication gate [8,26]. This is because each party needs to send the shares representing its local multiplication to every other party, thus requiring $\mathcal{O}(n^2)$ messages, and hence $\mathcal{O}(n^2)$ bits if we only look at complexity depending on n.

Even if we look at such protocols in the pre-processing model, where the so-called "Beaver multiplication triples" are produced in an offline phase [4], and we are primarily concerned about the communication complexity of the online phase, a similar situation occurs. In such protocols, see for example [19], 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 $\mathcal{O}(n^2)$ communication complexity.

In the SPDZ protocol [22], for the case of *non-robust*[4] maliciously secure MPC (with abort) in the dishonest majority setting (i.e. with $t < n$), an online communication complexity of $\mathcal{O}(n)$ 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 $2 \cdot n$

[4] An MPC protocol is called robust if the honest parties obtain the correct output at the end of the protocol irrespective of the behaviour of the corrupted parties, otherwise it is called non-robust.

messages being sent, and is hence $\mathcal{O}(n)$. 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. Our goal is to achieve such a result for robust protocols in the *pre-processing model*.

Related Work: With $t < n/3$, information-theoretically secure an online protocols with $\mathcal{O}(n)$ communication per multiplication are presented in [21]. There the basic idea is a new method of reconstructing a batch of $\Theta(n)$ secret-shared values with $\mathcal{O}(n^2)$ 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 $\mathcal{O}(n)$ *online complexity* is needed. In [21] a similar protocol in the pre-processing model is also sketched, which uses the designatred reconstructor idea (similar to the idea used in SPDZ, discussed above). The protocol is only sketched, and appears to require $O(t)$ rounds to identify the faulty shares; as opposed to our method which requires no additional rounds.

In [28], a computationally-secure MPC protocol with $t < n/2$ and communication complexity $\mathcal{O}(n)$ 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 to 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. 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 online methods are easily applicable to the asynchronous communication setting[5], which models real-world networks like the Internet more appropriately than the synchronous communication setting.

In [9], an information-theoretically secure MPC protocol in the pre-processing model with $t < n/2$ and $\mathcal{O}(n)$ communication complexity per multiplication is presented. Both the offline and online phase of [9] are designed in the dispute control framework [5], which is a generalisation of the player-elimination technique and so like other papers in the same framework it is not known if the protocol can be made to work in the more practical asynchronous communication setting. Moreover since their online phase protocol is in the dispute control framework, it requires $\mathcal{O}(n^2 + \mathcal{D})$ rounds of interaction in the online phase, where $\mathcal{D}$ is the multiplicative depth of the circuit. This is unlike other MPC protocols in the pre-processing model whose online phase requires only $\mathcal{O}(\mathcal{D})$ rounds of interaction [21,6,10,22]. Our technique for the online phase protocol does not deploy any player-elimination/dispute-control techniques and so requires fewer rounds than [9]. And our online phase can be executed even in the asynchronous setting with $t < n/2$ and $\mathcal{O}(n)$ communication complexity. Imagine a scenario involving a large number of parties, participating from various parts of the globe. Clearly (an asynchronous) online protocol with less number of communication rounds is desirable here and so our online phase protocol will fit the bill appropriately. In the non-preprocessing model, information-theoretically secure MPC protocols with "near linear" amortized communication complexity but *non-optimal resilience* are presented in [3,20,25]. Namely the overall communication complexity of these protocols are $\mathcal{O}\left(\text{polylog}(n, C) \cdot C\right)$, where C is the circuit size. While the protocol of [20] is perfectly-secure and can tolerate upto $t < (1/3 - \epsilon) \cdot n$ corruptions where $0 < \epsilon < 1/3$, the protocols in [3,25] are statistical with resilience $t < (1/2 - \epsilon) \cdot n$ where $0 < \epsilon < 1/2$. The central idea in these protocols is to take advantage of the non-optimal resilience by deploying packed secret-sharing, where "several" values are secret shared simultaneously via a single sharing instance. None of the protocols are known to work in asynchronous settings and all of them heavily rely on the fact that there are more honest parties than just $1/2$ (making them non-optimal in terms of resilience).

[5] We stress that we are interested only in the online complexity. Unlike our online phase, our offline phase protocol cannot be executed in a completely asynchronous setting with $t < n/2$.

Finally we note that an asynchronous MPC protocol with $t < n/3$ and $\mathcal{O}(n)$ communication complexity in the preprocessing model is presented in [17]. However the online phase protocol of [17] is based on the $\mathcal{O}(n)$ reconstruction method of [6,21] with $t < n/3$ and hence cannot be adapted to the $t < n/2$ setting.

Our Contribution: We present a computationally-secure method to obtain $\mathcal{O}(n)$ communication complexity 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 [19], BDOZ [10] and SPDZ [22]. Our main contribution is a trick to robustly reconstruct a secret with an amortized communication complexity of $\mathcal{O}(n)$ messages. 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. To detect the error positions we apply the the pair-wise BDOZ MACs from [10]. 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.

Before proceeding we pause to examine the communication complexity of the offline phase of protocols such as SPDZ. 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 or BDOZ style MACs. And this is what we exactly do to implement our offline phase in the synchronous setting. In [22] the offline communication complexity is given as $\mathcal{O}(n^2/s)$ 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 [23], 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 $\mathcal{O}(n)$ in terms of the number of messages. However, each message is $\mathcal{O}(s)$ and so the bit communication complexity is still $\mathcal{O}(n^2)$.

As our online phase also works in the asynchronous setting, we explore how the offline phase, and the interaction between the offline and online phases can be done asynchronously. For this we follow the VIFF framework [19], which implements the offline phase asynchronously with $t < n/3$ via the pseudo-random secret sharing, assuming a single synchronization point between the offline and online phases. Following the same approach, we show how the interaction between our offline and online phase can be handled asynchronously with $t < n/2$. However we require an additional technicality for $t < n/2$ to deal with the issue of agreement among the parties at the end of asynchronous offline phase. Specifically, we either require "few" synchronous rounds or a non-equivocation mechanism at the end of offline phase to ensure agreement among the parties. We stress that once this is done then the online phase protocol can be executed in a completely asynchronous fashion with $t < n/2$.

2 Preliminaries

We assume a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, connected by pair-wise authentic channels, and a centralized static, active PPT adversary $\mathcal{A}$ who can corrupt any $t < n/2$ parties. For simplicity we assume $n = 2t + 1$, so that $t = \Theta(n)$. The functionality that the parties wish to compute is represented by an arithmetic circuit over a finite field $\mathbb{F}$, where $|\mathbb{F}| > n$. We denote by μ and κ the statistical and cryptographic security parameter respectively. A negligible function in κ (μ) will be denoted by $\mathsf{negl}(\kappa)$ ($\mathsf{negl}(\mu)$), while $\mathsf{negl}(\kappa, \mu)$ denotes a function which is negligible in both κ and μ. We use both information-theoretic and public-key cryptographic primitives in our protocols. The security of the information theoretic primitives are parameterised with μ, while that of cryptographic primitives are parametrised with κ. We assume $\mathbb{F} = \mathrm{GF}(p)$, where p is a prime with $p \approx 2^\mu$, to ensure that the statistical security of our protocol holds with all but $\mathsf{negl}(\mu)$ probability. Each element of $\mathbb{F}$ can be represented by μ bits. For vectors $A = (a_1, \ldots, a_m)$ and $B = (b_1, \ldots, b_m)$, $A \otimes B$ denotes the value $\sum_{i=1}^{m} a_i b_i$. The ith element in a vector A is denoted as $A[i]$ and (i, j)th element in a matrix A as $A[i, j]$.

2.1 Communication Settings

In this paper we consider two 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: A party first performs some computation, then they send messages to the others parties over the pair-wise channels and broadcast any message which need to be broadcast; this stage is followed by receiving both the messages sent to the party by the other parties over the pair-wise channels and the messages broadcast by the other parties. 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) 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 here 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 [7,12], 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 hears 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). We assume an IND-CPA secure linearly-homomorphic public-key encryption scheme set-up for every $P_i \in \mathcal{P}$ with message space $\mathbb{F}$; a possible instantiation could be the BGV scheme [11]. Under this set-up, P_i will own a secret decryption key $\mathbf{dk}^{(i)}$ and the corresponding encryption key $\mathbf{pk}^{(i)}$ will be publicly known. Given $\mathbf{pk}^{(i)}$, a plaintext x and a randomness r, anyone can compute a ciphertext $\mathsf{HE}.\mathbf{c}(x) \overset{def}{=} \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(x, r)$ of x for P_i, using the encryption algorithm HE.Enc, where the size of $\mathsf{HE}.\mathbf{c}(x)$ is $\mathcal{O}(\kappa)$ bits. Given a ciphertext $\mathsf{HE}.\mathbf{c}(x) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(x, \star)$ and the decryption key $\mathbf{dk}^{(i)}$, P_i can recover the plaintext $x = \mathsf{HE}.\mathsf{Dec}_{\mathbf{dk}^{(i)}}(\mathbf{c}_x)$ using the decryption algorithm HE.Dec. The encryption scheme is assumed to be *linearly homomorphic*: given two ciphertexts $\mathsf{HE}.\mathbf{c}(x) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(x, \star)$ and $\mathsf{HE}.\mathbf{c}(y) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(y, \star)$, there exists an operation, say $\oplus$, such that $\mathsf{HE}.\mathbf{c}(x) \oplus \mathsf{HE}.\mathbf{c}(y) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(x + y, \star)$. Moreover, given a ciphertext $\mathsf{HE}.\mathbf{c}(x) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(x, \star)$ and a public constant c, there exists some operation, say $\odot$, such that $c \odot \mathsf{HE}.\mathbf{c}(x) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(c \cdot x, \star)$.

Information-theoretic MACs: We will use information-theoretically secure MAC, similar to the one used in [10]. Here a random pair $\mathsf{K} = (\alpha, \beta) \in \mathbb{F}^2$ is selected as the MAC key and the MAC tag on a value $a \in \mathbb{F}$, under the key K is defined as $\mathsf{MAC}_{\mathsf{K}}(a) \overset{def}{=} \alpha \cdot a + \beta$. The MACs will be used as follows: a party P_i will hold some value a and a MAC tag $\mathsf{MAC}_{\mathsf{K}}(a)$, while party P_j will hold the MAC key K. Later when P_i wants to disclose a to P_j, it sends

a along with $\mathsf{MAC}_{\mathsf{K}}(a)$; P_j verifies if a is consistent with the MAC tag with respect to its key K. A *corrupted* party P_i on holding the MAC tag on a message gets one point on the straight-line $y = \alpha x + \beta$ and it leaves one degree of freedom on the polynomial. Therefore even a computationally unbounded P_i cannot recover K completely. So a corrupted P_i cannot reveal an incorrect value $a' \neq a$ to an honest P_j without getting caught, except with probability $\frac{1}{|\mathbb{F}|} \approx 2^{-\mu} = \mathsf{negl}(\mu)$, which is the probability of guessing a second point on the straight-line. We call two MAC keys $\mathsf{K} = (\alpha, \beta)$ and $\mathsf{K}' = (\alpha', \beta')$ *consistent* if $\alpha = \alpha'$. Given two consistent MAC keys $\mathsf{K} = (\alpha, \beta)$ and $\mathsf{K}' = (\alpha, \beta')$ and a public constant c, we define the following operations on MAC keys:

$$\mathsf{K} + \mathsf{K}' \overset{def}{=} (\alpha, \beta + \beta'), \quad \mathsf{K} + c \overset{def}{=} (\alpha, \beta + \alpha c) \quad \text{and} \quad c \cdot \mathsf{K} \overset{def}{=} (\alpha, c \cdot \beta).$$

Given two consistent MAC keys K, K' and a value c, the following *linearity* properties hold for the MAC:

- **Addition:**

$$\mathsf{MAC}_{\mathsf{K}}(a) + \mathsf{MAC}_{\mathsf{K}'}(b) = \mathsf{MAC}_{\mathsf{K}+\mathsf{K}'}(a + b).$$

- **Addition/Subtraction by a Constant:**

$$\mathsf{MAC}_{\mathsf{K}-c}(a + c) = \mathsf{MAC}_{\mathsf{K}}(a) \quad \text{and} \quad \mathsf{MAC}_{\mathsf{K}+c}(a - c) = \mathsf{MAC}_{\mathsf{K}}(a).$$

- **Multiplication by a constant:**

$$c \cdot \mathsf{MAC}_{\mathsf{K}}(a) = \mathsf{MAC}_{c \cdot \mathsf{K}}(c \cdot a).$$

2.3 The Various Sharings

We define following two types of secret sharing.

Definition 1 **($[\cdot]$-sharing).** *We say a value $s \in \mathbb{F}$ is $[\cdot]$-shared among $\mathcal{P}$ if there exists a polynomial $p(\cdot)$ of degree at most t with $p(0) = s$ and every (honest) party $P_i \in \mathcal{P}$ 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 $\mathcal{P}$. That is, $[s] = \{s_i\}_{i=1}^{n}$.*

Definition 2 **($\langle\cdot\rangle$-sharing).** *We say that a value $s \in \mathbb{F}$ is $\langle\cdot\rangle$-shared among $\mathcal{P}$ if s is $[\cdot]$-shared among $\mathcal{P}$ 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 $P_i, P_j \in \mathcal{P}$: party P_i holds MAC tag $\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i)$ for a MAC key K_{ji} held by P_j. We denote by $\langle s\rangle$ the vector of such shares, MAC keys and MAC tags of s corresponding to the (honest) parties in $\mathcal{P}$. That is, $\langle s\rangle = \big\{s_i, \{\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i), \mathsf{K}_{ij}\}_{j=1}^{n}\big\}_{i=1}^{n}$.*

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

Definition 3 **($\langle\langle\cdot\rangle\rangle_j$-sharing).** *Let $s \in \mathbb{F}$ and $[s] = \{s_i\}_{i=1}^{n}$ be the vector of shares corresponding to an $[\cdot]$-sharing of s. We say that s is $\langle\langle\cdot\rangle\rangle_j$-shared among $\mathcal{P}$ with respect to a designated party P_j, if every (honest) party P_i holds an encrypted share $\mathsf{HE}.\mathbf{c}(s_i)$ and encrypted MAC tag $\mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i))$ under the public key $\mathbf{pk}^{(j)}$, such that P_j holds the MAC keys K_{ji} and the secret key $\mathbf{dk}^{(j)}$. We denote by $\langle\langle s\rangle\rangle_j$ the vector of encrypted shares and encrypted MAC tags corresponding to the (honest) parties in $\mathcal{P}$, along with the MAC keys and the secret key of P_j. That is, $\langle\langle s\rangle\rangle_j = \Big\{\{\mathsf{HE}.\mathbf{c}(s_i), \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i))\}_{i=1}^{n}, \{\mathsf{K}_{ji}\}_{i=1}^{n}, \mathbf{dk}^{(j)}\Big\}$.*

Private Reconstruction of $\langle\cdot\rangle$ and $\langle\langle\cdot\rangle\rangle$-shared Value Towards a Designated Party. Note that with $n = 2t + 1$, a $[\cdot]$-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 $\langle\cdot\rangle$-sharing towards a designated party, say P_j, by asking the parties to send their shares, along with MAC tags to P_j, who then identifies the correct shares with high probability and reconstructs the secret. A similar idea can be used to reconstruct an $\langle\langle s\rangle\rangle_j$-sharing towards P_j. Now the parties send encrypted shares and MAC tags to P_j, who decrypts them before doing the verification. We call the resultant protocols $\mathsf{RecPrv}(\langle s\rangle, P_j)$ and $\mathsf{RecPrvEnc}(\langle\langle s\rangle\rangle_j)$ respectively, which are presented in Fig. 1. We stress that while $\langle s\rangle$ can be reconstructed towards *any* P_j, $\langle\langle s\rangle\rangle_j$ can be reconstructed only towards P_j, as P_j alone holds the secret key $\mathbf{dk}^{(j)}$ that is required to decrypt the shares and the MAC tags.

Protocol $\mathsf{RecPrv}(\langle s\rangle, P_j)$

- Every party $P_i \in \mathcal{P}$ sends its share s_i and the MAC tag $\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i)$ to the party P_j.
- Party P_j on receiving the share s'_i and the MAC tag $\mathsf{MAC}'_{\mathsf{K}_{ji}}(s_i)$ from P_i computes $\mathsf{MAC}_{\mathsf{K}_{ji}}(s'_i)$ and verifies if $\mathsf{MAC}_{\mathsf{K}_{ji}}(s'_i) \overset{?}{=} \mathsf{MAC}'_{\mathsf{K}_{ji}}(s_i)$.
- If the verification passes then P_j considers s'_i as a valid share.
- Once $t + 1$ valid shares are obtained, using them P_j interpolates the sharing polynomial and outputs its constant term as s.

Protocol $\mathsf{RecPrvEnc}(\langle\langle s\rangle\rangle_j)$

- Every party $P_i \in \mathcal{P}$ sends $\mathsf{HE}.\mathbf{c}(s_i)$ and $\mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i))$ to the party P_j.
- Party P_j, on receiving these values, computes $s'_i = \mathsf{HE}.\mathsf{Dec}_{\mathbf{dk}^{(j)}}(\mathsf{HE}.\mathbf{c}(s_i))$ and $\mathsf{MAC}'_{\mathsf{K}_{ji}}(s_i) = \mathsf{HE}.\mathsf{Dec}_{\mathbf{dk}^{(j)}}$ $(\mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(s_i)))$. The rest of the steps are the same as for $\mathsf{RecPrv}(\star, P_j)$.

Fig. 1. Protocols for Reconstructing a $\langle\cdot\rangle$-sharing and $\langle\langle\cdot\rangle\rangle$-sharing Towards a Designated Party

It is easy to see that if P_j is honest, then P_j correctly reconstructs the shared value in protocol RecPrv as well as in $\mathsf{RecPrvEnc}$, except with probability at most $\frac{t}{|\mathbb{F}|} \approx \mathsf{negl}(\mu)$. While protocol RecPrv has communication complexity $\mathcal{O}(\mu \cdot n)$ bits, protocol $\mathsf{RecPrvEnc}$ has communication complexity $\mathcal{O}(\kappa \cdot n)$ bits. Also note that both the protocols will work in the asynchronous setting. We argue this for RecPrv (the same argument will work for $\mathsf{RecPrvEnc}$). 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 RecPrv. Similar statements hold for protocol $\mathsf{RecPrvEnc}$. Thus we have the following Lemmas.

Lemma 1. *Let s be $\langle\cdot\rangle$-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* $\mathsf{negl}(\mu)$*, an honest P_j reconstructs s.*
- *Communication Complexity: The communication complexity is $\mathcal{O}(\mu \cdot n)$ bits.*

Lemma 2. *Let s be $\langle\cdot\rangle$-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 4 (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.*

Sharings $\langle\langle a\rangle\rangle_j$ and $\langle\langle b\rangle\rangle_j$ with respect to a designated P_j are called key-consistent if 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 $[\cdot]$-sharings: Given $[a] = \{a_i\}_{i=1}^n$ and $[b] = \{b_i\}_{i=1}^n$ and a public constant c, we have:

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

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

- *Addition*: To compute $\langle a+b\rangle$, every party P_i needs to locally compute $a_i + b_i$, $\{\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i) + \mathsf{MAC}_{\mathsf{K}'_{ji}}(b_i)\}_{j=1}^n$ and $\{\mathsf{K}_{ij} + \mathsf{K}'_{ij}\}_{j=1}^n$,
$$\langle a\rangle + \langle b\rangle = \langle a+b\rangle = \left\{a_i + b_i, \{\mathsf{MAC}_{\mathsf{K}_{ji}+\mathsf{K}'_{ji}}(a_i+b_i), \mathsf{K}_{ij} + \mathsf{K}'_{ij}\}_{j=1}^n\right\}_{i=1}^n .$$
- *Addition by a Public Constant*: To compute $\langle c+a\rangle$, every party P_i needs to locally compute $c + a_i$, In addition recall that $\mathsf{MAC}_{\mathsf{K}_{ji}-c}(a_i + c) = \mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)$. Hence we assign $\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)$ to $\mathsf{MAC}_{\mathsf{K}_{ji}-c}(a_i + c)$ and compute $\{\mathsf{K}_{ij} - c\}_{j=1}^n$.
$$c + \langle a\rangle = \langle c+a\rangle = \left\{c + a_i, \{\mathsf{MAC}_{\mathsf{K}_{ji}-c}(a_i + c), \mathsf{K}_{ij} - c\}_{j=1}^n\right\}_{i=1}^n .$$
- *Multiplication by a Public Constant*: To compute $\langle c \cdot a\rangle$, every party P_i needs to locally compute $c \cdot a_i$, $\{c \cdot \mathsf{MAC}_{\cdot\mathsf{K}_{ji}}(a_i)\}_{j=1}^n$ and $\{c \cdot \mathsf{K}_{ij}\}_{j=1}^n$,
$$c \cdot \langle a\rangle = \langle c \cdot a\rangle = \left\{c \cdot a_i, \{\mathsf{MAC}_{c\cdot\mathsf{K}_{ji}}(c \cdot a_i), c \cdot \mathsf{K}_{ij}\}_{j=1}^n\right\}_{i=1}^n .$$

Linearity of $\langle\langle\cdot\rangle\rangle_j$-sharings: Given $\langle\langle a\rangle\rangle_j = \left\{\{\mathsf{HE}.\mathbf{c}(a_i), \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)),\}_{i=1}^n, \ \{\mathsf{K}_{ji}\}_{i=1}^n, \mathbf{dk}^{(j)}\right\}$ and $\langle\langle b\rangle\rangle_j = \left\{\{\mathsf{HE}.\mathbf{c}(b_i), \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(b_i))\}_{i=1}^n, \{\mathsf{K}'_{ji}\}_{i=1}^n, \mathbf{dk}^{(j)}\right\}$ that are key-consistent we can add the sharings via the operation

$$\begin{aligned}\langle\langle a\rangle\rangle_j + \langle\langle b\rangle\rangle_j &= \langle\langle a+b\rangle\rangle_j \\ &= \Big\{\{\mathsf{HE}.\mathbf{c}(a_i + b_i), \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}+\mathsf{K}'_{ji}}(a_i + b_i))\}_{i=1}^n, \\ &\qquad \{\mathsf{K}_{ji} + \mathsf{K}'_{ji}\}_{i=1}^n, \mathbf{dk}^{(j)}\Big\}\end{aligned}$$

So to compute $\langle\langle a+b\rangle\rangle_j$, every party $P_i \in \mathcal{P}$ needs to locally compute the values $\mathsf{HE}.\mathbf{c}(a_i) \oplus \mathsf{HE}.\mathbf{c}(b_i)$ and $\mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)) \oplus \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{\mathsf{K}'_{ji}}(b_i))$, while party P_j needs to compute $\{\mathsf{K}_{ji} + \mathsf{K}'_{ji}\}_{i=1}^n$.

Generating $\langle\langle\cdot\rangle\rangle_j$-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 $\mathsf{HE}.\mathbf{c}(r)$ of r, under the public key $\mathbf{pk}^{(j)}$, and a $\langle\cdot\rangle$-sharing $\langle a\rangle = \left\{a_i, \{\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i), \mathsf{K}_{ij}\}_{j=1}^n\right\}_{i=1}^n$. Given the above, the parties need to compute a $\langle\langle\cdot\rangle\rangle_j$ sharing:

$$\langle\langle r \cdot a\rangle\rangle_j = \left\{\{\mathsf{HE}.\mathbf{c}(r \cdot a_i), \mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{r\cdot\mathsf{K}_{ji}}(r \cdot a_i))\}_{i=1}^n, \{r \cdot \mathsf{K}_{ji}\}_{i=1}^n, \mathbf{dk}^{(j)}\right\}$$

of $r \cdot a$. Computing the above needs only local computation by the parties. Specifically, each party $P_i \in \mathcal{P}$ locally computes the values $\mathsf{HE}.\mathbf{c}(r \cdot a_i) = a_i \odot \mathsf{HE}.\mathbf{c}(r)$ and

$$\mathsf{HE}.\mathbf{c}(\mathsf{MAC}_{r\cdot\mathsf{K}_{ji}}(r \cdot a_i)) = \mathsf{HE}.\mathbf{c}(r \cdot \mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)) = \mathsf{MAC}_{\mathsf{K}_{ji}}(a_i) \odot \mathsf{HE}.\mathbf{c}(r),$$

since $r \cdot \mathsf{MAC}_{\mathsf{K}_{ji}}(a_i) = \mathsf{MAC}_{r\cdot\mathsf{K}_{ji}}(a_i \cdot r)$. Finally party P_j locally computes $\{r \cdot \mathsf{K}_{ji}\}_{i=1}^n$.

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

We present a new protocol to publicly reconstruct $n(t+1)\frac{\kappa}{\mu} = \Theta(\frac{n^2\kappa}{\mu})$ $\langle\cdot\rangle$-shared values with communication complexity $\mathcal{O}(\kappa \cdot n^3)$ bits. So the amortized communication overhead for public reconstruction of one $\langle\cdot\rangle$-shared value is linear in n i.e. $\mathcal{O}(\mu \cdot n)$ bits. For a better understanding of the ideas used in the protocol, we first present a protocol RecPubSimple to publicly reconstruct $n(t+1)$ $\langle\cdot\rangle$-shared values with communication complexity $\mathcal{O}(\kappa \cdot n^3)$ bits. We will then extend this protocol for $n(t+1)\frac{\kappa}{\mu}$ secrets while retaining the same communication complexity; the resulting protocol is called RecPub.

Let $\{\langle a^{(i,j)}\rangle\}_{i=1,j=1}^{n,t+1}$ 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 RecPrv, where $\Theta(n^2)$ instances are run to reconstruct all the values to a single party. This method has communication complexity $\mathcal{O}(\kappa \cdot n^4)$ bits and thus has a quadratic overhead. Our approach outperforms the naive method, and 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 ith row of A; i.e. $A_i(x) \overset{def}{=} A[i,1] + A[i,2]x + \ldots, A[i,t+1]x^t$. Let B denote an $n \times n$ matrix and $B[i,j] \overset{def}{=} A_i(j)$, for $i,j \in \{1,\ldots,n\}$. 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 $\mathcal{O}(\kappa \cdot n^3)$ bits. In what follows, we denote ith row and column of A as A_i and A^i respectively, with a similar notation used for the rows and columns of B.

Since B_i is linearly dependent on A_i, given $\langle\cdot\rangle$-sharing of A_i, it requires only local computation to generate $\langle\cdot\rangle$-sharings of the elements in B_i. Specifically, $\langle B[i,j]\rangle = \langle A[i,1]\rangle + \langle A[i,2]\rangle \cdot j + \ldots + \langle A[i,t+1]\rangle \cdot j^t$. Then we reconstruct the elements of A to all the parties in two steps. First B^i is reconstructed towards P_i using n instances of RecPrv with an overall cost $\mathcal{O}(\mu \cdot n^3)$ bits. Next each party P_i sends B^i to all the parties, requiring $\mathcal{O}(\mu \cdot n^3)$ 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 being 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 RecPubSimple (which can be done beforehand in the offline phase of the main MPC protocol). Every party P_j holds a secret key $\mathbf{dk}^{(j)}$ for the linearly-homomorphic encryption scheme HE and the corresponding public key $\mathbf{pk}^{(j)}$ is publicly available. In addition, P_j holds a vector R^j of n random combiners and the encryptions $\mathsf{HE}.\mathbf{c}(R^j[1]),\ldots,\mathsf{HE}.\mathbf{c}(R^j[n])$ of the values in R^j under P_j's public key $\mathbf{pk}^{(j)}$ are available publicly. The above setup can be created once and for all, and can be reused across multiple instances of RecPubSimple.

Given the above random combiners in an encrypted form, party P_j can obtain the linear combination $c^{(i,j)} \overset{def}{=} \sum_{l=1}^{n} B^i[l]R^j[l]$ of the elements of B^i as follows. First note that the parties hold $\langle\cdot\rangle$-sharing of the elements of B^i. If the linear combiners were publicly known, then the parties could compute $\langle c^{(i,j)}\rangle = \sum_{l=1}^{n} R^j[l]\langle B^i[l]\rangle$ 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 PKE. Specifically, given encryptions $\mathsf{HE}.\mathbf{c}(R^j[l])$ under $\mathbf{pk}^{(j)}$ and sharings $\langle B^i[l]\rangle$, the parties first generate $\langle\langle R^j[l] \cdot B^i[l]\rangle\rangle_j$ for every P_j (recall that it requires only local computation). Next the parties linearly combine the sharings $\langle\langle R^j[l] \cdot B^i[l]\rangle\rangle_j$ for $l = 1,\ldots,n$ to obtain $\langle\langle c^{(i,j)}\rangle\rangle_j$, which is then reconstructed towards party P_j using an instance of RecPrvEnc. In total n^2 such instances need to be executed, costing $\mathcal{O}(\kappa \cdot n^3)$ bits. Protocol RecPubSimple is presented in Fig. 2.

Protocol $\mathsf{RecPubSimple}(\{\langle a^{(i,j)}\rangle\}_{i=1,j=1}^{n,t+1})$

Each $P_j \in \mathcal{P}$ holds R^j and the encryptions $\mathsf{HE}.\mathbf{c}(R^j[1]), \ldots, \mathsf{HE}.\mathbf{c}(R^j[n])$, under P_j's public key $\mathbf{pk}^{(j)}$, are publicly known. Let A be the matrix of size $n \times (t+1)$, with (i,j)th entry as $a^{(i,j)}$, for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, t+1\}$. We denote the ith row and column of A as A_i and A^i respectively. Let $A_i(x) \stackrel{def}{=} a^{(i,1)} + \ldots + a^{(i,t+1)}x^t$ for $i \in \{1, \ldots, n\}$. Let B be the matrix of size $n \times n$, with the (i,j)th entry as $B[i,j] \stackrel{def}{=} A_i(j)$ for $i, j \in \{1, \ldots, n\}$. We denote the ith row and column of B as B_i and B^i respectively. The parties do the following to reconstruct A:

- **Computing $\langle\cdot\rangle$-sharing of every element of B:** For $i, j \in \{1, \ldots, n\}$, the parties compute $\langle B[i,j]\rangle = \langle A[i,1]\rangle + j \cdot \langle A[i,1]\rangle + \ldots + j^t \cdot \langle A[i,t+1]\rangle$.
- **Reconstructing B^i towards P_i:** For $i \in \{1, \ldots, n\}$, the parties execute $\mathsf{RecPrv}(\langle B[1,i]\rangle, P_i), \ldots, \mathsf{RecPrv}(\langle B[n,i]\rangle, P_i)$ 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 $c^{(i,j)} \stackrel{def}{=} B^i \otimes R^j$:
  - The parties first compute $\langle\langle R^j[l] \cdot B^i[l]\rangle\rangle_j$ from $\mathsf{HE}.\mathbf{c}(R^j[l])$ and $\langle B^i[l]\rangle$ for $l \in \{1, \ldots, n\}$ and then compute $\langle\langle c^{(i,j)}\rangle\rangle_j = \sum_{l=1}^{n} \langle\langle R^j[l] \cdot B^i[l]\rangle\rangle_j$.
  - The parties execute $\mathsf{RecPrvEnc}(\langle\langle c^{(i,j)}\rangle\rangle_j)$ to reconstruct $c^{(i,j)}$ towards P_j.
- **Sending B^i to all:** Each $P_i \in \mathcal{P}$ sends B^i to every $P_j \in \mathcal{P}$. Each P_j then reconstructs A as follows:
  - On receiving $\bar{B}^i$ from P_i, compute $c'^{(i,j)} = \bar{B}^i \otimes R^j$ and check if $c^{(i,j)} \stackrel{?}{=} c'^{(i,j)}$. If the test passes then P_j considers $\bar{B}^i$ 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 $\langle\cdot\rangle$-shared Values with $\mathcal{O}(\kappa \cdot n)$ Communication Complexity

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 Appendix A, in conjunction with the online phase of our MPC protocol.

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

– *Correctness: Except with probability* $\mathsf{negl}(\kappa, \mu)$*, every honest party reconstructs $\{a^{(i,j)}\}_{i=1,j=1}^{n,t+1}$.*
– *Communication Complexity: The communication complexity is $\mathcal{O}(\kappa \cdot n^3)$ bits.*

From $\mathcal{O}(\kappa \cdot n)$ to $\mathcal{O}(\mu \cdot n)$ Amortized cost of Reconstruction. We note that the amortized complexity of reconstructing one secret via $\mathsf{RecPubSimple}$ is $\mathcal{O}(\kappa \cdot n)$, where κ is the cryptographic security parameter. To improve the amortized cost to $\mathcal{O}(\mu \cdot n)$, we make the following observation on the communication in $\mathsf{RecPubSimple}$. There is a scope to amortize part of the communication to reconstruct more than $n(t+1)$ secrets. This leads to a trick that brings down the amortized communication complexity per secret to $\mathcal{O}(\mu \cdot n)$ bits. We call our new protocol RecPub. which starts with $\frac{\kappa}{\mu}$ batches of secrets where each batch consists of $n(t+1)$ secrets. For each batch, RecPub executes exactly the same steps as done in $\mathsf{RecPubSimple}$ except for the step involving the reconstruction of $B^i \otimes R^j$. RecPub keeps the communication cost of this step unperturbed by taking a random linear combination of $\frac{\kappa}{\mu}$ B^i columns together. Therefore RecPub still needs private reconstruction of n^2 $\langle\langle\cdot\rangle\rangle_j$-shared values and a communication of $\mathcal{O}(\kappa \cdot n^3)$ bits for this step. For the rest of the steps, the communication complexity of RecPub will be $\frac{\kappa}{\mu}$ times the communication complexity of the same steps in $\mathsf{RecPubSimple}$. Since $\mathsf{RecPubSimple}$ requires $\mathcal{O}(\mu \cdot n^3)$ bits of communication for the rest of the steps, the communication complexity of RecPub will turn out to be $\mathcal{O}(\kappa \cdot n^3)$ bits of communication overall. Since the number reconstructed secrets are $n(t+1)\frac{\kappa}{\mu}$, RecPub offers an amortized cost of $\mathcal{O}(\mu \cdot n)$ bits per secret. The formal specification of protocol RecPub is in Fig. 3.

We note that RecPub takes random linear combination of $\frac{\kappa n}{\mu}$ values. So the one time set up has to be enhanced where every P_j now holds $\frac{\kappa n}{\mu}$ random combiners, and the encryptions of them are available in public. Namely R^j is

Protocol $\mathsf{RecPub}(\{\langle a^{(i,j,k)}\rangle\}_{i=1,j=1,k=1}^{n,t+1,\kappa/\mu})$

Each $P_j \in \mathcal{P}$ holds $R^j = (R^j[1], \ldots, R^j[\kappa n/\mu])$ and the encryptions $\mathsf{HE}.\mathbf{c}(R^j[1]), \ldots, \mathsf{HE}.\mathbf{c}(R^j[\kappa n/\mu])$, under P_j's public key $\mathbf{pk}^{(j)}$, are publicly known. For a k that varies over $1, \ldots, \kappa/\mu$, the set of secrets $\{a^{(i,j,k)}\}_{i=1,j=1}^{n,t+1}$ is denoted as the kth batch of secrets. Let $\mathcal{A}^{(k)}$ be the matrix of size $n \times (t+1)$ consisting of the kth batch i.e., the (i,j)th entry of $\mathcal{A}^{(k)}$ is $a^{(i,j,k)}$, for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, t+1\}$. We denote the ith row and column of $\mathcal{A}^{(k)}$ as $\mathcal{A}_i^{(k)}$ and ${\mathcal{A}^{(k)}}^i$ respectively. We define $\mathcal{A}_i^{(k)}(x) \overset{def}{=} \mathcal{A}^{(k)}[i,1] + \mathcal{A}^{(k)}[i,2] \cdot x + \ldots + \mathcal{A}^{(k)}[i,t+1] \cdot x^t$ for $i \in \{1, \ldots, n\}$. Let $\mathcal{B}^{(k)}$ be the matrix of size $n \times n$, with the (i,j)th entry as $\mathcal{B}^{(k)}[i,j] \overset{def}{=} \mathcal{A}_i^{(k)}(j)$ for $i, j \in \{1, \ldots, n\}$. We denote the ith row and column of $\mathcal{B}^{(k)}$ as $\mathcal{B}_i^{(k)}$ and ${\mathcal{B}^{(k)}}^i$ respectively. We denote the concatenation of the ith column of all the $\mathcal{B}^{(k)}$s as B^i i.e. $B^i = \left[({\mathcal{B}^{(1)}}^i)^{tr}, \ldots, ({\mathcal{B}^{(\frac{\kappa}{\mu})}}^i)^{tr}\right]$ where $(\cdot)^{tr}$ denotes vector transpose. The parties do the following to reconstruct $\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(\frac{\kappa}{\mu})}$:

- **Computing $\langle\cdot\rangle$-sharing of elements of $\mathcal{B}^{(k)}$ for $k = 1, \ldots, \kappa/\mu$:** Same as in $\mathsf{RecPubSimple}$.
- **Reconstructing ${\mathcal{B}^{(k)}}^i$ towards P_i for $k = 1, \ldots, \kappa/\mu$:** Same as in $\mathsf{RecPubSimple}$. Party P_i holds B^i at the end of this step.
- **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 $c^{(i,j)} \overset{def}{=} B^i \otimes R^j = \sum_{l=1}^{\kappa n/\mu} R^j[l]B^i[l]$:
  - The parties first compute $\langle\langle R^j[l]B^i[l]\rangle\rangle_j$ from $\mathsf{HE}.\mathbf{c}(R^j[l])$ and $\langle B^i[l]\rangle$ for $l \in \{1, \ldots, \frac{\kappa n}{\mu}\}$ and then compute $\langle\langle c^{(i,j)}\rangle\rangle_j = \sum_{l=1}^{\frac{\kappa n}{\mu}} \langle\langle R^j[l]B^i[l]\rangle\rangle_j$.
  - The parties execute $\mathsf{RecPrvEnc}(\langle\langle c^{(i,j)}\rangle\rangle_j)$ to reconstruct $c^{(i,j)}$ towards P_j.
- **Sending B^i 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 $\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(\frac{\kappa}{\mu})}$ as follows:
  - On receiving $\bar{B}^i$ from P_i, compute $c'^{(i,j)} = \bar{B}^i \otimes R^j$ and check if $c^{(i,j)} \overset{?}{=} c'^{(i,j)}$. If the test passes then P_j interprets $\bar{B}^i$ as $\left[({\mathcal{B}^{(1)}}^i)^{tr}, \ldots, ({\mathcal{B}^{(\frac{\kappa}{\mu})}}^i)^{tr}\right]$ and considers ${\mathcal{B}^{(k)}}^i$ as the valid i^{th} column of the matrix $\mathcal{B}^{(k)}$ for $k = 1, \ldots, \kappa/\mu$.
  - For $k = 1, \ldots, \kappa/\mu$, once $t+1$ valid columns of $\mathcal{B}^{(k)}$ are obtained by P_j, it then reconstructs $\mathcal{A}^{(k)}$.

Fig. 3. Robustly Reconstructing $\langle\cdot\rangle$-shared Values with $\mathcal{O}(\mu \cdot n)$ Communication Complexity

a vector of $\kappa n/\mu$ random values and the encryptions $\mathsf{HE}.\mathbf{c}(R^j[1]), \ldots, \mathsf{HE}.\mathbf{c}(R^j[\kappa n/\mu])$ done under P_j's public key $\mathbf{pk}^{(j)}$ are available publicly. We thus have the following Lemma.

Lemma 4. *Let $\{\langle a^{(i,j,k)}\rangle\}_{i=1,j=1,k=1}^{n,t+1,\kappa/\mu}$ be a set of $n(t+1)\frac{\kappa}{\mu}$ shared values which need to be publicly reconstructed by the parties. Then given a setup $(\mathbf{pk}^{(1)}, \mathbf{dk}^{(1)}), \ldots, (\mathbf{pk}^{(n)}, \mathbf{dk}^{(n)})$ for the linearly-homomorphic encryption scheme* HE *for the n parties and encryptions $\mathsf{HE}.\mathbf{c}(R^j[1]), \ldots, \mathsf{HE}.\mathbf{c}(R^j[\kappa n/\mu])$ of $\kappa n/\mu$ random values in R^j on the behalf of each party $P_j \in \mathcal{P}$, with only P_j knowing the random values, protocol* RecPub *achieves the following in the synchronous communication setting:*

- *Correctness: Except with probability* $\mathsf{negl}(\kappa, \mu)$*, every honest party reconstructs $\{a^{(i,j,k)}\}_{i=1,j=1,k=1}^{n,t+1,\kappa/\mu}$.*
- *Communication Complexity: The communication complexity is $\mathcal{O}(\kappa \cdot n^3)$ bits.*

Protocol $\mathsf{RecPubSimple}$ **and** RecPub **in the Asynchronous Setting:** Consider the protocol $\mathsf{RecPubSimple}$, and note that the steps involving interaction among the parties are during the instances of RecPrv and $\mathsf{RecPrvEnc}$. All the remaining steps involve only local computation by the parties. As the instances of RecPrv and $\mathsf{RecPrvEnc}$ eventually terminate for each honest party, it follows that $\mathsf{RecPubSimple}$ eventually terminates for each honest party in the asynchronous setting. Similar arguments hold for RecPub, so we get the following lemma.

Lemma 5. *Protocol* RecPub *achieves the following in the asynchronous communication setting:*

- *Correctness & Communication Complexity: Same as in Lemma 4*
- *Termination: Every honest party eventually terminates the protocol.*

4 Linear Overhead Online Phase Protocol

Let $f : \mathbb{F}^n \rightarrow \mathbb{F}$ be a publicly known function over $\mathbb{F}$, represented as an arithmetic circuit C over $\mathbb{F}$, consisting of M multiplication gates. Then using our efficient reconstruction protocol RecPub enables one to securely realize the standard ideal functionality $\mathcal{F}_f$ (see Fig. 4 for an explicit functionality) for the MPC evaluation of the circuit C, in the $\mathcal{F}_{\textsc{Prep}}$-hybrid model, with communication complexity $\mathcal{O}(\mu \cdot (n \cdot M + n^2))$ bits, thus providing a linear overhead per multiplication gate. More specifically, assume that the parties have access to an ideal pre-processing and input processing functionality $\mathcal{F}_{\textsc{Prep}}$, which creates the following *one-time* setup: **(i)** Every P_j holds a secret key $\mathbf{dk}^{(j)}$ for the linearly-homomorphic encryption scheme HE and the corresponding public key $\mathbf{pk}^{(j)}$ is available publicly. In addition, each P_j holds n random combiners $R^{(j)} = (r^{(j,1)}, \ldots, r^{(j,n)})$ and the encryptions $\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j,n)})$ of these values under P_j's public key are publicly available. **(ii)** Each P_i holds α_{ij}, the α-component of all its keys for party P_j (recall that for key-consistent sharings every P_i has to use the same α-component for all its keys corresponding to P_j). The above setup can be reused across multiple instances of $\mathit{\Pi}_{\textsc{Online}}$ and can be created once and for all. In addition to the one-time setup, the functionality also creates at least M random $\langle\cdot\rangle$-shared multiplications triples (these are not reusable and have to be created afresh for every execution of $\mathit{\Pi}_{\textsc{Online}}$) and $\langle\cdot\rangle$-shared inputs of the parties. Functionality $\mathcal{F}_{\textsc{Prep}}$ is presented in Fig. 5. In $\mathcal{F}_{\textsc{Prep}}$, 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.

Functionality $\mathcal{F}_f$

$\mathcal{F}_f$ interacts with the parties $P_1, \ldots, P_n$ and the adversary $\mathcal{S}$ and is parametrized by an n-input function $f : \mathbb{F}^n \rightarrow \mathbb{F}$ represented as an arithmetic circuit C.

- Upon receiving $(i, x^{(i)})$ from every party $P_i \in \mathcal{P}$ where $x^{(i)} \in \mathbb{F}$, the functionality computes $y = C(x^{(1)}, \ldots, x^{(n)})$, sends y to all the parties and the adversary $\mathcal{S}$ and halts. Here C denotes the arithmetic circuit over $\mathbb{F}$ representing f.

Fig. 4. The Ideal Functionality for Computing a Given Function

Using $\mathcal{F}_{\textsc{Prep}}$ we design a protocol $\mathit{\Pi}_{\textsc{Online}}$ (see Fig. 6) which realizes $\mathcal{F}_f$ in the synchronous setting and provides *universal composability* (UC) security [13,10,22,16]. The protocol is based on the standard Beaver's idea of securely evaluating the circuit in a shared fashion using pre-processed shared random multiplication triples [4] 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 $\mathcal{F}_{\textsc{Prep}}$ and evaluate the multiplication gate by using Beaver's trick. Specifically, let $\langle p\rangle, \langle q\rangle$ 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 $\mathcal{F}_{\textsc{Prep}}$, 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 $d \overset{def}{=} p - a$ and $e \overset{def}{=} q - b$. So if d and e are publicly known then $\langle p \cdot q\rangle = d \cdot e + d \cdot \langle b\rangle + e \cdot \langle a\rangle + \langle c\rangle$ 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.

To achieve the linear overhead in $\mathit{\Pi}_{\textsc{Online}}$, we require that the circuit is "wide" in the sense that at every level there are at least $n(t+1)\frac{\kappa}{\mu}$ independent multiplication gates that can be evaluated in parallel. This is to ensure that we can use our linear-overhead reconstruction protocol RecPub. We note that similar restrictions are used in some of the previous MPC protocols to achieve a linear overhead in the online phase. For example, [21,6,15] requires $\mathit{\Theta}(n)$ independent multiplication gates at each level to ensure that they can use their linear-overhead reconstruction protocol to evaluate these gates. In practice many functions have such a level of parallel multiplication gates when expressed

Functionality $\mathcal{F}_{\text{PREP}}$

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 $\{\mathbf{pk}^{(i)}, \mathbf{dk}^{(i)}\}_{i=1}^{n}$ of the linearly-homomorphic encryption scheme HE,
  - For each P_i, it selects $\frac{\kappa n}{\mu}$ random values $R^i = (r^{(i,1)}, \ldots, r^{(i, \frac{\kappa n}{\mu})})$, computes $\mathsf{HE}.\mathbf{c}(r^{(i,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(i, \frac{\kappa n}{\mu})})$ under $\mathbf{pk}^{(i)}$,
  - To party P_i it sends

$$\left(\mathbf{dk}^{(i)}, (r^{(i,1)}, \ldots, r^{(i, \frac{\kappa n}{\mu})}), \{\mathbf{pk}^{(j)}\}_{j=1}^{n}, \{\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j, \frac{\kappa n}{\mu})})\}_{j=1}^{n}\right).$$

  - On the behalf of each honest $P_i \in \mathcal{P} \setminus \mathcal{C}$, it selects n random values $\{\alpha_{ij}\}_{j=1}^{n}$, where the jth value is designated to be used in the MAC key for party P_j. On the behalf of each corrupted party $P_i \in \mathcal{C}$, it receives from $\mathcal{S}$ the α_{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 $\mathsf{Triples}$ from all the parties in $\mathcal{P}$, the functionality generates $\langle\cdot\rangle$-sharing of χ 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 $(x^{(i)}, i, \mathsf{Input})$ from party P_i and (i, 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 $\{\beta_{ij}\}_{P_i \in \mathcal{C},\ P_j \notin \mathcal{C}}$ 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 $\mathsf{K}_{ij} = (\alpha_{ij}, \beta_{ij})$ where α_{ij} was specified by $\mathcal{S}$ in 'Setup generation' stage. It computes the MAC tag $\mathsf{MAC}_{\mathsf{K}_{ij}}(s_j)$ of s_j for every honest P_j corresponding to the key of every corrupted P_i.
- On receiving MAC tags $\{\mathsf{MAC}_{ij}\}_{P_i \in \mathcal{C},\ P_j \notin \mathcal{C}}$ 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_j corresponding to P_i as $\mathsf{K}_{ji} = (\alpha_{ji}, \beta_{ji})$ where $\beta_{ji} = \mathsf{MAC}_{ij} - \alpha_{ji} \cdot s_i$ and α_{ji} 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 $\mathsf{K}_{jk} = (\alpha_{jk}, \beta_{jk})$ where α_{jk} is taken from 'setup generation phase' and β_{jk} is chosen randomly. It then computes the corresponding MAC tag of P_k as $\mathsf{MAC}_{\mathsf{K}_{jk}}(s_k)$.
- It sends $\left\{s_j, \{\mathsf{MAC}_{\mathsf{K}_{kj}}, \mathsf{K}_{jk}\}_{k=1}^{n}\right\}$ 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

in arithmetic circuit format, and practical systems use algorithms to maximise the level of such parallelism in their execution, see e.g. [32].

The properties of Π_{ONLINE} are stated in Theorem 1, which is proved in Appendix A. In the protocol, $2M$ $\langle\cdot\rangle$-shared values are publicly reconstructed via RecPub while evaluating the multiplication gates. Assuming that the M multiplication gates can be divided into blocks of $n(t+1)\frac{\kappa}{\mu}$ independent multiplication gates, evaluating the same will cost $\mathcal{O}(\kappa n^3 \cdot \frac{\mu M}{n(t+1)\kappa}) = \mathcal{O}(\mu \cdot n \cdot M)$ bits. The only steps in Π_{ONLINE} which require interaction among the parties are during the instances of the reconstruction protocols, which eventually terminate for the honest parties. Hence we get Theorem 2 for the asynchronous setting.

Theorem 1. *Protocol Π_{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 $\mathcal{O}(\mu \cdot (n \cdot M + n^2))$ bits.*

Theorem 2. *Protocol Π_{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 $\mathcal{O}(\mu \cdot (n \cdot M + n^2))$ bits.*

Protocol $\mathit{\Pi}_{\text{ONLINE}}$

Every party $P_i \in \mathcal{P}$ interact with $\mathcal{F}_{\text{PREP}}$ with input Setup, $\mathsf{Triples}$ and $(x^{(i)}, i, \mathsf{Input})$ and receives $\mathbf{dk}^{(i)}$, $\{\mathbf{pk}^{(j)}\}_{j=1}^{n}$, $R^i = (r^{(i,1)}, \ldots, r^{(i, \frac{\kappa n}{\mu})})$, $\{\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j, \frac{\kappa n}{\mu})})\}_{j=1}^{n}$, its information for multiplication triples $\{(\langle a^{(l)} \rangle, \langle b^{(l)} \rangle, \langle c^{(l)} \rangle)\}_{l=1}^{M}$ and its information for inputs $\{\langle x^{(j)} \rangle\}_{j=1}^{n}$. The honest parties associate the sharing $(\langle a^{(l)} \rangle, \langle b^{(l)} \rangle, \langle c^{(l)} \rangle)$ with the l^{th} multiplication gate for $l \in \{1, \ldots, M\}$ 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 an $\langle \cdot \rangle$-sharing of the gate output.
- **Multiplication Gates**: M multiplication gates as grouped as a batch of $n \cdot (t+1) \cdot \frac{\kappa}{\mu}$. We explain the evaluation for one batch. Let the inputs to the ith batch be $\{(\langle p^{(l)} \rangle, \langle q^{(l)} \rangle)\}_{l=1}^{n \cdot (t+1) \cdot \frac{\kappa}{\mu}}$ and let $\{(\langle a^{(l)} \rangle, \langle b^{(l)} \rangle, \langle c^{(l)} \rangle)\}_{l=1}^{n \cdot (t+1) \cdot \frac{\kappa}{\mu}}$ be the corresponding associated multiplication triples. To compute $\langle p^{(l)} \cdot q^{(l)} \rangle$, the parties do the following:
  - Locally compute $\langle d^{(l)} \rangle = \langle p^{(l)} \rangle - \langle a^{(l)} \rangle = \langle p^{(l)} - a^{(l)} \rangle$ and $\langle e^{(l)} \rangle = \langle q^{(l)} \rangle - \langle b^{(l)} \rangle_t = \langle q^{(l)} - b^{(l)} \rangle$.
  - Publicly reconstruct the values $\{d^{(l)}\}_{l=1}^{n \cdot (t+1) \cdot \frac{\kappa}{\mu}}$ and $\{e^{(l)}\}_{l=1}^{n \cdot (t+1) \cdot \frac{\kappa}{\mu}}$ using two instances of RecPub.
  - On reconstructing $d^{(l)}, e^{(l)}$, the parties set $\langle p^{(l)} \cdot q^{l)} \rangle = d^{(l)} \cdot e^{(l)} + d^{(l)} \cdot \langle b^{(l)} \rangle + e^{(l)} \cdot \langle a^{(l)} \rangle + \langle c^{(l)} \rangle$.
- **Output Gate**: Let $\langle y \rangle$ be the sharing of the output gate. The parties execute $\mathsf{RecPrv}(\langle y \rangle, P_i)$ for every $P_i \in \mathcal{P}$, robustly reconstruct y and terminate.

Fig. 6. Realizing $\mathcal{F}_f$ with a Linear Overhead in $\mathcal{F}_{\text{PREP}}$-hybrid Model for the Synchronous Setting

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

Securely Realizing $\mathcal{F}_{\text{PREP}}$ in the Synchronous Setting. In Appendix B, we present a protocol $\mathit{\Pi}_{\text{PREP}}$ which realizes $\mathcal{F}_{\text{PREP}}$ in the synchronous setting and achieves UC security. The protocol is a straight forward adaptation of the offline phase protocol of [10,22] to deal with Shamir sharing, instead of additive sharing.

Securely Realizing $\mathcal{F}_{\text{PREP}}$ with Abort in the Partial Synchronous Setting. Any secure realization of $\mathcal{F}_{\text{PREP}}$ has to ensure that all the honest parties have an *agreement* on the final outcome, which is impossible in the asynchronous setting with $t < n/2$ [29,30]. Another difficulty in realizing $\mathcal{F}_{\text{PREP}}$ in an asynchronous setting is that it is possible to ensure input provision from only $n - t$ parties to avoid endless wait. For $n = 2t + 1$ this implies that there may be only one honest input provider. This may not be acceptable for most practical applications of MPC. To get rid of the latter difficulty, [19] introduced the following 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 C we examine how to make $\mathit{\Pi}_{\text{PREP}}$ 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 non-equivocation mechanism (which can be implemented using a trusted hardware module).

Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grants EP/I03126X and EP/M016803, by DARPA and the US Navy under contract #N66001-15-C-4070, and by the Infosys Foundation.

References

1. G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.
2. M. Backes, F. Bendun, A. Choudhury, and A. Kate. Asynchronous MPC with a Strict Honest Majority Using Non-equivocation. In M. M. Halldórsson and S. Dolev, editors, *PODC*, pages 10–19. ACM, 2014.
3. J. Baron, K. E. Defrawy, J. Lampkins, and R. Ostrovsky. How to Withstand Mobile Virus Attacks, Revisited. In M. M. Halldórsson and S. Dolev, editors, *PODC*, pages 293–302. ACM, 2014.
4. D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer Verlag, 1991.
5. Z. Beerliová-Trubíniová and M. Hirt. Efficient Multi-party Computation with Dispute Control. In S. Halevi and T. Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328. Springer, 2006.
6. Z. Beerliová-Trubíniová and M. Hirt. Perfectly-Secure MPC with Linear Communication Complexity. In R. Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer Verlag, 2008.
7. M. Ben-Or, R. Canetti, and O. Goldreich. Asynchronous Secure Computation. In S. R. Kosaraju, D. S. Johnson, and A. Aggarwal, editors, *STOC*, pages 52–61. ACM, 1993.
8. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In J. Simon, editor, *STOC*, pages 1–10. ACM, 1988.
9. E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680. Springer, 2012.
10. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
11. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):13:1–13:36, 2014.
12. R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute, Israel, 1995.
13. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 13(1):143–202, 2000.
14. D. Chaum, C. Crépeau, and I. Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *STOC*, pages 11–19. ACM, 1988.
15. A. Choudhury, M. Hirt, and A. Patra. Asynchronous Multiparty Computation with Linear Communication Complexity. In Y. Afek, editor, *DISC*, volume 8205 of *Lecture Notes in Computer Science*, pages 388–402. Springer, 2013.
16. A. Choudhury, J. Loftus, E. Orsini, A. Patra, and N. P. Smart. Between a Rock and a Hard Place: Interpolating between MPC and FHE. In K. Sako and P. Sarkar, editors, *ASIACRYPT*, volume 8270, pages 221–240. Springer, 2013.
17. A. Choudhury and A. Patra. Optimally Resilient Asynchronous MPC with Linear Communication Complexity. In S. K. Das, D. Krishnaswamy, S. Karkar, A. Korman, M. Kumar, M. Portmann, and S. Sastry, editors, *ICDCN*, pages 5:1–5:10. ACM, 2015.
18. A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. On the (Limited) Power of Non-equivocation. In D. Kowalski and A. Panconesi, editors, *PODC*, pages 301–308. ACM, 2012.
19. I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In S. Jarecki and G. Tsudik, editors, *PKC*, pages 160–179, 2009.
20. I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly Secure Multiparty Computation and the Computational Overhead of Cryptography. In H. Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.
21. I. Damgård and J. B. Nielsen. Scalable and Unconditionally Secure Multiparty Computation. In A. Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590. Springer Verlag, 2007.
22. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
23. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security - ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
24. M. Fitzi and M. Hirt. Optimally Efficient Multi-valued Byzantine Agreement. In E. Ruppert and D. Malkhi, editors, *PODC*, pages 163–168. ACM Press, 2006.
25. D. Genkin, Y. Ishai, and A. Polychroniadou. Efficient Multi-party Computation: From Passive to Active Security via Secure SIMD Circuits. In R. Gennaro and M. Robshaw, editors, *CRYPTO*, volume 9216 of *Lecture Notes in Computer Science*, pages 721–741. Springer, 2015.

26. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In B. A. Coan and Y. Afek, editors, *podc*, pages 101–111. ACM, 1998.
27. O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
28. M. Hirt and J. B. Nielsen. Robust Multiparty Computation with Linear Communication Complexity. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer, 2006.
29. M. Hirt, J. B. Nielsen, and B. Przydatek. Cryptographic Asynchronous Multi-party Computation with Optimal Resilience (Extended Abstract). In *EUROCRYPT*, LNCS 3494, pages 322–340. Springer Verlag, 2005.
30. M. Hirt, J. B. Nielsen, and B. Przydatek. Asynchronous Multi-Party Computation with Quadratic Communication. In *ICALP*, LNCS 5126, pages 473–485. Springer Verlag, 2008.
31. J. Katz and C. Y. Koo. On Expected Constant-Round Protocols for Byzantine Agreement. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 445–462. Springer, 2006.
32. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS'13*, pages 549–560. ACM, 2013.
33. A. C. Yao. Protocols for Secure Computations (Extended Abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

A Proof of Theorem 1

We refer to [10,13,16,22] for the definition of UC-security. Let $\mathcal{A}$ be a real-world adversary corrupting t parties during the execution of $\mathit{\Pi}_{\text{ONLINE}}$ and let $\mathcal{C} \subset \mathcal{P}$ denote the set of corrupted parties. We present a simulator $\mathcal{S}_f$ for $\mathcal{A}$, who interacts with $\mathcal{F}_f$ and simulates each received message of $\mathcal{A}$ in the protocol $\mathit{\Pi}_{\text{ONLINE}}$ from the honest parties and from the functionality $\mathcal{F}_{\text{PREP}}$, stage by stage. Note that the simulator will also simulate the steps of the subprotocol RecPub executed inside the protocol $\mathit{\Pi}_{\text{ONLINE}}$. We present the high level idea of the simulator first, for formal details see Figures 7 and 8.

The idea of $\mathcal{S}_f$ is straight forward. The simulator plays the role of $\mathcal{F}_{\text{PREP}}$ and honestly simulates the call to $\mathcal{F}_{\text{PREP}}$. Specifically, on receiving Setup from the corrupted parties, it creates n public key, secret key pairs $\{\mathbf{pk}^{(i)}, \mathbf{dk}^{(i)}\}_{i=1}^{n}$ of the linearly-homomorphic encryption scheme HE, selects $\frac{\kappa n}{\mu}$ random values $R^j = (r^{(j,1)}, \ldots, r^{(j,\frac{\kappa n}{\mu})})$ and computes their encryptions $\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j,\frac{\kappa n}{\mu})})$ under $\mathbf{pk}^{(j)}$ and sends

$$\Big(\mathbf{dk}^{(i)}, (r^{(i,1)}, \ldots, r^{(i,\frac{\kappa n}{\mu})}), \{\mathbf{pk}^{(j)}\}_{j=1}^{n}, \{\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j,\frac{\kappa n}{\mu})})\}_{j=1}^{n}\Big)$$

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

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 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 RecPub protocol steps (for the latter, recall that an honest P_j is supposed to test if $c'^{(i,j)} = \bar{B}^i \otimes R^j \overset{?}{=} c^{(i,j)}$ 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 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 $\mathcal{F}_f$. Since the simulator knows the simulated output sharing, say $\langle\widetilde{y}\rangle$, it can easily create a "fake" output sharing $\langle y\rangle$ such that the shares of the corrupted parties in $\langle\widetilde{y}\rangle$ 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 $\langle \widetilde{y} \rangle$. 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 $\mathcal{A}$ corresponding to the honest parties. Using these shares, $\mathcal{A}$ 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 $\langle \cdot \rangle$-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. corresponding to the honest parties. The reason is that the simulator knows in advance $\langle \cdot \rangle_{ij}$ for every corrupted $P_i \in \mathcal{C}$ and honest $P_j \notin \mathcal{C}$ for each of the sharing involved in the computation. That is, it knows in advance the information that corrupted P_i is supposed to send to an honest P_j either for MAC test or for linear combiner test. If the adversary sends wrong information for the above verifications and still passes the tests, then the simulator halts. Whereas in the hybrid world, the honest parties will carry on the computation as the tests pass and may end up outputting wrong output. However the probability with which the simulated world and the hybrid would will differ in this regard is the same as the probability with which the adversary can pass the MAC test or the random combiner test corresponding to the MAC keys and the random combiners of the honest parties. For the MAC test, the probability is same as the probability of forging a MAC tag and is negligible. For the random combiner test, the probability is the same as the chances of the adversary in guessing the random combiners from their encrypted form. So the security here can be reduced to the security of the underlying HE scheme. In other words, if the HE is semantically secure, then the probability of guessing the random linear combiners of an honest party is negligible. This implies that the simulated and the hybrid views are indistinguishable from the view point of the environment. This completes our proof.

Simulator $\mathcal{S}_f$ – Part 1

Let HE be a linearly-homomorphic encryption scheme. The simulator plays the role of the honest parties and simulates each step of the protocol Π_{ONLINE} as follows. The communication of the environment $\mathcal{Z}$ with the adversary $\mathcal{A}$ is handled as follows: Every input value received by the simulator from $\mathcal{Z}$ 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 $\mathcal{Z}$). 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 $\mathcal{F}_{\mathbf{PREP}}$: $\mathcal{S}_f$ 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 $\{(\langle a^{(l)} \rangle), \langle b^{(l)} \rangle), \langle c^{(l)} \rangle)\}_{l=1}^{\chi}$, input sharings $\{\langle x^{(i)} \rangle\}_{P_i \in \mathcal{C}}$ of the corrupted parties and the inputs sharings $\{\widetilde{\langle x^{(i)} \rangle}\}_{P_i \in \mathcal{P} \setminus \mathcal{C}}$ corresponding to honest (we denote the input of an honest party P_i with $\widetilde{x^{(i)}}$ since it is not the real input of P_i): (i) $\langle \cdot \rangle_j$ for every honest P_j and (ii) $\langle \cdot \rangle_{ij}$ for every corrupted $P_i \in \mathcal{C}$ and honest party P_j. It also knows the combiners $R^i = r^{(i,1)}, \ldots, r^{(i, \frac{\kappa n}{\mu})}$ and the encrypted combiners $\mathsf{HE}.\mathbf{c}(r^{(i,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(i, \frac{\kappa n}{\mu})})$ for each $P_i \in \mathcal{P}$, along with the public-key secret-key pairs $\{(\mathbf{pk}^{(i)}, \mathbf{dk}^{(i)})\}_{i=1}^{n}$.

Fig. 7. Simulator for the adversary $\mathcal{A}$ corrupting t parties in the set $\mathcal{C} \subset \mathcal{P}$ during Π_{ONLINE} – Part 1

Simulator $\mathcal{S}_f$ – Part 2

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 $n \cdot (t+1) \cdot \frac{\kappa}{\mu}$. We show the simulation for one batch. Let $\{(\langle\widetilde{p^{(l)}}\rangle, \langle\widetilde{q^{(l)}}\rangle)\}_{l=1}^{n\cdot(t+1)\cdot\frac{\kappa}{\mu}}$ be the pair of sharings corresponding to the input pairs of the $n \cdot (t+1) \cdot \frac{\kappa}{\mu}$ multiplication gates which need to be evaluated. Moreover, let $\{(\langle a^{(l)}\rangle, \langle b^{(l)}\rangle, \langle c^{(l)}\rangle)\}_{l=1}^{n\cdot(t+1)\cdot\frac{\kappa}{\mu}}$ be the sharing of the corresponding associated multiplication triples. Corresponding to the sharings $\langle\widetilde{d^{(l)}}\rangle = \langle\widetilde{p^{(l)}}\rangle - \langle a^{(l)}\rangle = \langle\widetilde{p^{l)}} - a^{(l)}\rangle$ and $\langle\widetilde{e^{(l)}}\rangle = \langle\widetilde{q^{(i,j)}}\rangle - \langle b^{(l)}\rangle = \langle\widetilde{q^{(l)}} - b^{(l)}\rangle$, the simulator computes: (i) $\langle\cdot\rangle_j$ for every honest P_j 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 RecPub to reconstruct $\{\widetilde{d^{(l)}}, \widetilde{e^{(l)}}\}_{l=1}^{n\cdot(t+1)\cdot\frac{\kappa}{\mu}}$ towards the corrupted parties in $\mathcal{C}$. On receiving the messages from the parties in $\mathcal{C}$ in 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 abort. Finally, corresponding to the following sharings, $\langle\widetilde{p^{(l)}} \cdot \widetilde{q^{(l)}}\rangle = \widetilde{d^{(l)}}\widetilde{e^{(l)}} + \widetilde{d^{(l)}}\langle\widetilde{b^{(l)}}\rangle + \widetilde{e^{(l)}}\langle\widetilde{a^{(l)}}\rangle + \langle\widetilde{c^{(l)}}\rangle$, the simulator computes: (i) $\langle\cdot\rangle_j$ for every honest P_j 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 $(i, x^{(i)})$ and obtains the function output y. Let $\langle\widetilde{y}\rangle$ be the sharing associated with the output gate. $\mathcal{S}_f$ at this stage knows $\langle\widetilde{y}\rangle_j$ for each honest P_j and $\langle\widetilde{y}\rangle_{ij}$ of each corrupted P_i and honest P_j. The simulator then simulates the messages of the honest parties corresponding to the instances of $\mathsf{RecPrv}(\langle\widetilde{y}\rangle, \star)$ as follows:
  - Let $\{\widetilde{y}_i\}_{P_i\in\mathcal{C}}$ be the shares of the corrupted parties corresponding to $[\widetilde{y}]$. Using these shares and the value y, the simulator generates the simulated output sharing $[y]$, such that the shares of the corrupted parties $P_i \in \mathcal{C}$ are same in $[y]$ and $[\widetilde{y}]$.
  - 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 $\mathsf{K}_{ij}^{(\widetilde{y})}$, the MAC key of P_i, corresponding to each honest P_j in the sharing $\langle\widetilde{y}\rangle$. Moreover, let y_j be the share of y for P_j in the sharing $[y]$. It then computes the simulated MAC tag $\mathsf{MAC}_{\mathsf{K}_{ji}^{(\widetilde{y})}}(y_j)$ for the share y_j under the MAC key $\mathsf{K}_{ij}^{(\widetilde{y})}$.
  - 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 $\mathsf{MAC}_{\mathsf{K}_{ji}^{(\widetilde{y})}}(y_j)$ to $\mathcal{A}$ as part of $\mathsf{RecPrv}(\langle\widetilde{y}\rangle, \star)$, 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. 8. Simulator for the adversary $\mathcal{A}$ corrupting t parties in the set $\mathcal{C} \subset \mathcal{P}$ during $\mathit{\Pi}_{\text{ONLINE}}$ – Part 2

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

We present a protocol $\mathit{\Pi}_{\text{PREP}}$ 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. We first discuss the existing primitives and a setup functionality that we require in $\mathit{\Pi}_{\text{PREP}}$.

B.1 Primitives

For $\mathit{\Pi}_{\text{PREP}}$ we assume the following primitives.

Multi-valued Broadcast Protocol with a Linear Overhead: Given a public-key set-up, [24] presents a multi-valued broadcast protocol with $t < n/2$ and communication complexity of $\mathcal{O}(\ell \cdot n + n^4 \cdot (n + \kappa))$ bits to broadcast an ℓ-bit message. This implies that if $\ell = \Theta(n^3 \cdot (n + \kappa))$ then the protocol has communication complexity $\mathcal{O}(n \cdot \ell)$ bits, thus achieving a linear overhead.

Threshold Somewhat Homomorphic Encryption (SHE): In a threshold somewhat homomorphic scheme 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 SHE for our case, we can consider the threshold SHE schemes of [16,22]. In what follows, we give a high level discussion of the key features of the encryption scheme that we use.

The cryptosystem has message space $\mathbb{F}$, public encryption key pk and a secret decryption key dk, Shamir-shared among the parties with threshold t, with each party P_i holding a decryption-key share dk_i.

- Given pk, a plaintext $x \in \mathbb{F}$ and a randomness r, anyone can compute a ciphertext $\mathsf{SHE}.\mathbf{c}(x) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(x, r)$, using the encryption algorithm $\mathsf{SHE}.\mathsf{Enc}$, where the size of $\mathsf{SHE}.\mathbf{c}(x)$ is $\mathcal{O}(\kappa)$ bits.
- Given a ciphertext $\mathsf{SHE}.\mathbf{c}(x) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(x, \star)$, any party P_i can compute, using its decryption-key share dk_i, a *decryption share* $\mu_i = \mathsf{SHE}.\mathsf{ShareDec}_i(\mathsf{dk}_i, \mathsf{SHE}.\mathbf{c}(x))$ of $\mathsf{SHE}.\mathbf{c}(x)$, consisting of $\mathcal{O}(\kappa)$ bits; here $\mathsf{SHE}.\mathsf{ShareDec}_i$ is the "partial" decryption function for the party P_i.
- Finally, given at least $t+1$ "correct" decryption shares $\{\mu_i\}$ corresponding to $\mathsf{SHE}.\mathbf{c}(x)$, 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 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 $\mathsf{SHE}.\mathbf{c}(x)$ and[6] $\mathsf{SHE}.\mathbf{c}(y)$ under the same public key, there exists some operation on ciphertexts, say $\boxplus$, such that $\mathsf{SHE}.\mathbf{c}(x) \boxplus \mathsf{SHE}.\mathbf{c}(y) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(x+y)$. Moreover, given a ciphertext $\mathsf{SHE}.\mathbf{c}(x)$ and a publicly known constant c, using the linearly homomorphic property, one can compute $\mathsf{SHE}.\mathbf{c}(c \cdot x)$. One can homomorphically compute $\mathsf{SHE}.\mathbf{c}(x-y)$, given encryptions $\mathsf{SHE}.\mathbf{c}(x)$ and $\mathsf{SHE}.\mathbf{c}(y)$ under the same public key. Finally, the cryptosystem is *one-time multiplicatively homomorphic*. That is, given encryptions $\mathsf{SHE}.\mathbf{c}(x)$ and $\mathsf{SHE}.\mathbf{c}(y)$ (under the same public key), there exists some operation on ciphertexts, say $\boxdot$, such that $\mathsf{SHE}.\mathbf{c}(x) \boxdot \mathsf{SHE}.\mathbf{c}(y) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(x \cdot y)$. The operation $\boxdot$ can be applied at most *once*; a ciphertext that is obtained after applying $\boxdot$ 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 Plaintext Knowledge (PoPK) for** SHE: In this protocol, there exists a prover $P_i \in \mathcal{P}$, who computes encryptions of ℓ 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 $\mathcal{O}(\kappa \cdot n \cdot \ell)$ bits. This is achievable by combining the multi-valued broadcast protocol of [24] (provided ℓ is sufficiently large) and any 2-party non-interactive ZK (NIZK) protocol for PoPK for SHE (see for example [1,16,22]). The idea is the following: assume that the communication complexity of the 2-party NIZK protocol for PoPK is $\mathcal{O}(\kappa)$ bits; then P_i broadcasts the ℓ ciphertexts, along with the NIZK proof for each ciphertext (so total $\mathcal{O}(\kappa \cdot \ell)$ bits) via the multi-valued broadcast protocol of [24]. Assuming $\ell = \Theta(n^3 \cdot (n+\kappa))$, the total communication complexity becomes $\mathcal{O}(\kappa \cdot n \cdot \ell)$ 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 $\mathcal{O}(\kappa)$ bits (for example, see [22,1,16]).
- **Proof of Plaintext Knowledge (PoPK) for** HE: In this protocol, there exists a prover $P_i \in \mathcal{P}$ with public key $\mathbf{pk}^{(i)}$ who computes encryptions of ℓ values under $\mathbf{pk}^{(i)}$ 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 $\mathcal{O}(\kappa \cdot \ell)$ bits.

[6] We assume an implicit randomness used in a ciphertext until and unless it is explicitly stated.

- **Proof of Correct Multiplication (PoCM) for** HE: In this protocol, there exists a prover $P_j \in \mathcal{P}$ and a verifier $P_i \in \mathcal{P}$, with P_j and P_i knowing values α and a respectively. Moreover an encryption $\mathsf{HE}.\mathbf{c}(a)$ of a under the public key $\mathbf{pk}^{(i)}$ and an encryption $\mathsf{HE}.\mathbf{c}(\alpha)$ of α under the public key $\mathbf{pk}^{(j)}$ of HE are known to P_i and P_j. Prover P_j selects some β and homomorphically computes an encryption $\mathsf{HE}.\mathbf{c}(\gamma) = \alpha \odot \mathsf{HE}.\mathbf{c}(a) \oplus \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(\beta, \star)$ of $\gamma = \alpha \cdot a + \beta$ and sends $\mathsf{HE}.\mathbf{c}(\gamma)$ to P_i; note that $\mathsf{HE}.\mathbf{c}(\gamma)$ will be under the key $\mathbf{pk}^{(i)}$. Using this protocol, prover P_j can prove to P_i that it has computed $\mathsf{HE}.\mathbf{c}(\gamma)$ as above. The protocol has communication complexity $\mathcal{O}(\kappa)$ bits (see [10] for example).

Efficient Distributed Decryption with a Fewer Number of ZK Proofs: In protocol Π_{PREP}, we will require to decrypt several ciphertexts encrypted under the SHE scheme. 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 $\mathcal{O}(n^2)$ instances of PoCD for publicly decrypting a single ciphertext and hence $\mathcal{O}(n^2 \cdot \ell)$ such instances will be required to decrypt ℓ ciphertexts. Instead we borrow a protocol DistDec from [16], which overall requires $\mathcal{O}(n^3)$ instances of PoCD to decrypt ℓ 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 $t < n/2$ 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, an error will be detected for at most t instances, after which it will know all the t corrupted parties. So if a receiver requests a PoCD more than t times then definitely the receiver is corrupted. We note that during the protocol Π_{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 $\mathcal{O}(n^3)$, which is *independent* of the circuit size; for details see [16].

B.2 The Ideal One-time Setup Functionality

For Π_{PREP}, we assume an ideal set-up functionality $\mathcal{F}_{\text{ONE-TIME-SETUP}}$, presented in Figure 9. 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 HE 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, $\frac{\kappa n}{\mu}$ 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 α values are selected and given to it. The α values are designated to be used by that honest party as the α-component of the MAC key for different parties across all sharings to be used for a computation. The functionality creates encryptions of these α 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 α values which it wants to use as the α-component of the MAC keys corresponding to the honest parties. In addition, the functionality also receives the encryptions of these α values under the public key of the HE scheme of the corrupted party. If the encryptions are valid, then the functionality stores these α values and sends their encryptions to the respective honest parties.

Functionality $\mathcal{F}_{\text{ONE-TIME-SETUP}}$

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 $(\mathsf{pk}, \mathsf{dk})$ of the threshold SHE scheme SHE with threshold t, along with the secret-key shares $\mathsf{dk}_1, \ldots, \mathsf{dk}_n$ for the n parties. To every party $P_i \in \mathcal{P}$ it then sends $(\mathsf{pk}, \mathsf{dk}_i)$.
- **Creating Public/Secret keys of HE for Every Party**: It creates n public key, secret key pairs $\{\mathbf{pk}^{(i)}, \mathbf{dk}^{(i)}\}_{i=1}^{n}$ of the linearly-homomorphic encryption scheme HE and sends $\mathbf{dk}^{(i)}$ to party P_i for $i = 1, \ldots, n$ and sends $\{\mathbf{pk}^{(j)}\}_{j=1}^{n}$ 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 $R^j = (r^{(j,1)}, \ldots, r^{(j, \frac{\kappa n}{\mu})})$ and computes the encryptions $\mathsf{HE}.\mathbf{c}(r^{(j,i)}) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(j)}}(r^{(j,i)}, \star)$ for $i = 1, \ldots, \frac{\kappa n}{\mu}$. It then sends R^j to the party P_j and the encryptions $\mathsf{HE}.\mathbf{c}(r^{(j,1)}), \ldots, \mathsf{HE}.\mathbf{c}(r^{(j, \frac{\kappa n}{\mu})})$ 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 $\alpha_{i1}, \ldots, \alpha_{in}$ and computes the encryptions $\mathsf{HE}.\mathbf{c}(\alpha_{ij}) = \mathsf{HE}.\mathsf{Enc}_{\mathbf{pk}^{(i)}}(\alpha_{ij}, \star)$ for $j = 1, \ldots, n$. The functionality then sends $\{\alpha_{ij}, \mathsf{HE}.\mathbf{c}(\alpha_{ij})\}_{j=1}^{n}$ to P_i, along with the randomness used in the encryptions. To every party $P_j \in \mathcal{P}$, the functionality sends $\mathsf{HE}.\mathbf{c}(\alpha_{ij})$.
- **Sending Encrypted MAC Keys on the Behalf of Corrupted Parties Corresponding to Honest Parties**: On the behalf of every corrupted party $P_i \in \mathcal{C}$, the functionality receives from $\mathcal{S}$ the α_{ij} values that P_i would like to use as the α-component of the MAC key for each honest $P_j \in \mathcal{P} \setminus \mathcal{C}$. For each such α_{ij} value, the functionality also receives from $\mathcal{S}$ an encryption $\mathsf{HE}.\mathbf{c}(\alpha_{ij})$ of α_{ij} under the public key $\mathbf{pk}^{(i)}$. The functionality verifies if $\mathsf{HE}.\mathsf{Dec}_{\mathbf{dk}^{(i)}}(\mathsf{HE}.\mathbf{c}(\alpha_{ij})) \stackrel{?}{=} \alpha_{ij}$. If the verification passes then it stores α_{ij} on the behalf of P_i and sends the encryption $\mathsf{HE}.\mathbf{c}(\alpha_{ij})$ to the party P_j.

Fig. 9. Ideal Functionality for One-time Setup

B.3 Protocol Π_{PREP}

Protocol Π_{PREP} is presented in Figures 10 and 11. We present a high level overview of the protocol. The parties first call the functionality $\mathcal{F}_{\text{ONE-TIME-SETUP}}$ 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 χ triples, where $\chi \geq M + n$. We explain how one such random sharing $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ is generated; in the protocol the same steps are executed in parallel for χ batches and the broadcasts required for all the χ batches are done in parallel via the multi-valued broadcast protocol of [24].

The generation of $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ is done in two stages: in the first stage, the parties first generate $([a], [b], [c])$ and in the second stage, the parties generate pair-wise MACs to transform $([a], [b], [c])$ to $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$. To generate $([a], [b], [c])$, we use an idea similar to [22], extended for Shamir sharing. Specifically, to generate $[a]$, 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 $a_i \stackrel{def}{=} A(i)$ for every $P_i \in \{P_{t+2}, \ldots, P_n\}$, then clearly the vector $(a_1, \ldots, a_n)$ defines a sharing $[a]$ of the value $a \stackrel{def}{=} A(0)$. 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 $\{A(i)\}_{i=t+2}^{n}$ are publicly known linear combinations of the $t + 1$ values $\{a_i\}_{i=1}^{t+1}$. Since the values $\{a_i\}_{i=1}^{t+1}$ are available in an encrypted fashion, the parties can homomorphically compute encryptions of the $A(i)$ values corresponding to every $P_i \in \{P_{t+2}, \ldots, P_n\}$. Next these encryptions are decrypted *only* towards the party $P_{t+2}, \ldots, P_n$ respectively. The parties also homomorphically compute an encryption of a from the encryptions of $\{a_i\}_{i=1}^{t+1}$.

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.

Protocol $\Pi_{\textsc{Prep}}$ – Part 1

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

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

Generating Shared Multiplication Triples: The parties generate in parallel $\langle\cdot\rangle$-sharing of χ 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** $([a], [b], [c], [r])$ — 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 $\mathsf{SHE}.\mathbf{c}(a_i) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(a_i, \star), \mathsf{SHE}.\mathbf{c}(b_i) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(b_i, \star)$ and $\mathsf{SHE}.\mathbf{c}(r_i) = \mathsf{SHE}.\mathsf{Enc}_{\mathsf{pk}}(r_i, \star)$. It then broadcasts these encryptions, along with a ZK PoPK; the encryptions and the corresponding proofs for all the χ batches are broadcast together.
  - Let the set of $t+1$ values $\{(i, a_i)\}_{i=1}^{t+1}, \{(i, b_i)\}_{i=1}^{t+1}$ and $\{(i, r_i)\}_{i=1}^{t+1}$ define polynomials $A(\cdot), B(\cdot)$ and $R(\cdot)$ respectively. Define $a \overset{def}{=} A(0), b \overset{def}{=} B(0)$ and $r \overset{def}{=} R(0)$.
  - From the $t+1$ encryptions $\{\mathsf{SHE}.\mathbf{c}(a_i)\}_{i=1}^{t+1}$, the parties homomorphically compute the encryptions $\mathsf{SHE}.\mathbf{c}(a)$ and encryptions $\mathsf{SHE}.\mathbf{c}(A(t+2)), \ldots, \mathsf{SHE}.\mathbf{c}(A(n))$. Similarly, the parties homomorphically compute the encryptions $\mathsf{SHE}.\mathbf{c}(b)$ and encryptions $\mathsf{SHE}.\mathbf{c}(B(t+2)), \ldots, \mathsf{SHE}.\mathbf{c}(B(n))$ from the $t+1$ encryptions $\{\mathsf{SHE}.\mathbf{c}(b_i)\}_{i=1}^{t+1}$. In the same way, the parties homomorphically compute the encryptions $\mathsf{SHE}.\mathbf{c}(r)$ and encryptions $\mathsf{SHE}.\mathbf{c}(R(t+2)), \ldots, \mathsf{SHE}.\mathbf{c}(R(n))$ from the $t+1$ encryptions $\{\mathsf{SHE}.\mathbf{c}(r_i)\}_{i=1}^{t+1}$.
  - The parties homomorphically compute $\mathsf{SHE}.\mathbf{c}(c) = \mathsf{SHE}.\mathbf{c}(a) \boxdot \mathsf{SHE}.\mathbf{c}(b)$, followed by homomorphically computing $\mathsf{SHE}.\mathbf{c}(c+r) = \mathsf{SHE}.\mathbf{c}(c) \boxplus \mathsf{SHE}.\mathbf{c}(r)$.
  - The encryptions $\mathsf{SHE}.\mathbf{c}((A(t+2)), \mathsf{SHE}.\mathbf{c}(B(t+2)), \mathsf{SHE}.\mathbf{c}(R(t+2))), \ldots$ and $(\mathsf{SHE}.\mathbf{c}(A(n)), \mathsf{SHE}.\mathbf{c}(B(n)), \mathsf{SHE}.\mathbf{c}(R(n)))$ are decrypted (in a distributed fashion) towards party $P_{t+2}, \ldots, P_n$ respectively. In addition, the encryption $\mathsf{SHE}.\mathbf{c}(c+r)$ is distributed decrypted publicly. For doing these distributed decryptions, the parties execute the protocol $\mathsf{DistDec}$.
  - 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 $[c] = c + r - [r]$.

Fig. 10. Protocol for Realizing $\mathcal{F}_{\textsc{Prep}}$ in the $\mathcal{F}_{\textsc{One-time-Setup}}$-hybrid Model in the Synchronous Setting – Part 1

Specifically from encryptions of a, b, the parties homomorphically compute an encryption of $c = a \cdot b$, followed by homomorphically computing an encryption of $c + r$. The encryption of $c + r$ is publicly decrypted; since r is random and $[\cdot]$-shared, the privacy of c is maintained. Finally the parties set $[c] = (c + r) - [r]$.

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 $\mathsf{K}_{ji} = (\alpha_{ji}, \beta_{ji})$, held by party P_j. Note that if P_i is honest then an encryption $\mathsf{HE}.\mathbf{c}(\alpha_{ji})$ of α_{ji} under $\mathbf{pk}^{(j)}$ will be already known to P_i from $\mathcal{F}_{\textsc{One-time-Setup}}$. To compute the MAC tag, party P_i encrypts a_i under its public key $\mathbf{pk}^{(i)}$ and sends the encryption to P_j, along with a ZK PoPK. Party P_j then homomorphically computes an encryption of $\mathsf{MAC}_{\mathsf{K}_{ji}}(a_i)$ under $\mathbf{pk}^{(i)}$ 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_j uses consistent MAC keys for an honest P_i 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.

Protocol Π_{PREP} – Part 2

Generating Shared Multiplication Triples (cont'd):

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

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

Fig. 11. Protocol for Realizing $\mathcal{F}_{\text{PREP}}$ in the $\mathcal{F}_{\text{ONE-TIME-SETUP}}$-hybrid Model in the Synchronous Setting – Part 2

Note that the pair-wise MACs could be setup even by using the threshold SHE; but 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 $\mathcal{O}(n^2)$ pair-wise encrypted MAC tags for each sharing and decrypting one ciphertext via distributed decryption involves $\mathcal{O}(n)$ overhead.

In the protocol, each party needs to broadcast $\mathcal{O}(\chi)$ ciphertexts and NIZK proofs. If χ is sufficiently large then using the broadcast protocol of [24], this will cost in total $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi))$ bits. There will be $\mathcal{O}(\chi)$ ciphertexts which need to be publicly decrypted, costing $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi))$ bits of communication. In addition, $\mathcal{O}(n \cdot \chi)$ ciphertexts need to be decrypted towards designated parties, costing $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi))$ bits of communication. The decryptions (both public and private) are done via the protocol $\mathsf{DistDec}$ and in total $\mathcal{O}(n^3)$ instances of PoCD may be required, costing $\mathcal{O}(\kappa \cdot n^3)$ bits of communication. Finally setting up the pair-wise MACs will cost $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi))$ bits of communication. So the communication complexity of Π_{PREP} will be $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi + n^3))$ bits.

Since the protocol Π_{PREP} is a straight forward adaptation of the offline phase protocol of [10,22], we avoid giving the complete proof. Instead we state the following theorem.

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

C 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 $t < n/3$ [29,30]. 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, [19] 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 information to the adversary.

In [19] it was shown how to securely realize a variant of $\mathcal{F}_{\textsc{Prep}}$, augmented with abort (we call this functionality $\mathcal{F}_{\textsc{Prep-Abort}}$), in the partial asynchronous setting with $t < n/3$. The functionality $\mathcal{F}_{\textsc{Prep-Abort}}$ is similar to $\mathcal{F}_{\textsc{Prep}}$ 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 $\mathcal{F}_{\textsc{Prep-Abort}}$ distributes the information to the honest parties in the same way as done in $\mathcal{F}_{\textsc{Prep}}$, otherwise it sends $\perp$ to the honest parties.

To securely realize $\mathcal{F}_{\textsc{Prep-Abort}}$ in the partial asynchronous setting with $t < n/3$, [19] used the following approach: assume there exists a secure realization of $\mathcal{F}_{\textsc{Prep}}$, say $\Pi_{\textsc{Prep}}$, in a synchronous setting with broadcast (in [19], two instantiations of $\Pi_{\textsc{Prep}}$ are presented). Protocol $\Pi_{\textsc{Prep}}$ 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 $\mathcal{F}_{\textsc{Prep-Abort}}$ with $t < n/2$ in the partial asynchronous setting following the blueprint of [19], provided that we have the following two components: **(a)** A secure realization of $\mathcal{F}_{\textsc{Prep}}$ with $t < n/2$ in the synchronous communication setting with broadcast; **(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):

- *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 [24]. So if we assume $t + 1$ synchronous communication rounds after the synchronization point, then we can achieve agreement among the n parties on whether $\Pi_{\textsc{Prep}}$ when executed in the partial asynchronous setting, succeeded or not. Alternatively, one can assume a constant number of synchronous communication rounds after the synchronization point and run constant expected round synchronous agreement protocols [31].
- *Non-equivocation Mechanism*: In [18,2] 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 [18,2] 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 $\Pi_{\textsc{Prep}}$.

Now we discuss how to run the protocol $\Pi_{\textsc{Prep}}$ in the partial asynchronous setting following the method of [19]. The basic idea is to execute $\Pi_{\textsc{Prep}}$ 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 ith round (for instance, if the ith 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 $\Pi_{\textsc{Prep}}$ 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 $\Pi_{\textsc{Prep}}$ to 1 if *all* the following conditions hold. Otherwise it sets $q_i = 0$:

- P_i received all the messages it is supposed to receive in $\Pi_{\textsc{Prep}}$ before the timeout. In addition, P_i sent all the message that it is supposed to send in $\Pi_{\textsc{Prep}}$ before timeout.

- P_i's received broadcast messages in Π_{PREP} are the same as those received by all other parties.
- No instance of distributed decryption during the instances of $\mathsf{DistDec}$ in Π_{PREP} fails for the party P_i. Recall that in $\mathsf{DistDec}$ 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 $\mathsf{DistDec}$ in the partial asynchronous setting. Because if P_i detects any error then it stops participating in further rounds of Π_{PREP}. This ensures that every other honest party will stop participating in Π_{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 Π_{PREP} is successful and the parties proceed to execute the protocol Π_{ONLINE} with the values generated at the end of Π_{PREP}; otherwise the parties abort the protocol. The BA protocol ensures that if $q_i = 1$ for all the honest parties, then Π_{PREP} is successful and if $q_i = 0$ for all the honest parties then parties will abort Π_{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 Π_{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 Π_{PREP} when executed in the partial asynchronous setting will have communication complexity $\mathcal{O}(\kappa \cdot (n^2 \cdot \chi + \text{poly}(n)))$ bits. Moreover the protocol securely realizes $\mathcal{F}_{\text{PREP-ABORT}}$.